School of Net

code .education

Laravel 5 Essencial

Alta produtividade no mundo Real

Wesley Willians

schoolofnet.com | code.education

Laravel 5.1: Essencial

Alta produtividade no mundo Real

Wesley Willians Ramos da Silva

Esse livro está à venda em http://leanpub.com/laravel5essencial

Essa versão foi publicada em 2015-07-09

Dedico esse meu primeiro livro à minha princesa Sarah.

Conteúdo

Introdução

Sabemos que o mundo PHP está evoluindo exponencialmente, principalmente depois das diversas iniciativas de interoperabilidade como o PHP-FIG com suas PSR's e o surgimento do composer, que nos permite instalar quaisquer tipos de biblioteca de forma "plug 'n play" .

Com o ecosistema todo aquecido, onde o compartilhamento de bibliotecas e frameworks tornou-se algo extremamente simples, muitos projetos rapidamente começaram a ficar em evidência, incluindo o Laravel Framework.

Tenho que admitir que pela primeira vez que ouvi falar no Laravel, meu primeiro pensamento foi (e acredito que de muitos outros desenvolvedores também):

- Mais um framework PHP? Para que?
- Quais as vantagens que esse camarada pode trazer em cima de frameworks tão bem feitos como: Zend Framework, Symfony, etc.?

Admito que simplesmente ignorei o projeto.

Quem me conhece, sabe que sou uma pessoa extremamente focada e raramente fico divagando entre muitos projetos. Apesar de gostar de experimentar muito, é totalmente impossível ficar testando milhares de bibliotecas e frameworks o tempo todo.

Conforme o tempo passou, percebi que a cada dia muitos desenvolvedores começaram a falar mais e mais sobre o Lavarel, e nesse ponto resolvi instalar e dar uma brincada.

Quando fiz a instalação da versão 3 em meu computador, simplesmente fiquei extremamente desapontado. O Laravel me lembrava o Zend Framework 1, com muitas simplificações, e algumas má práticas de desenvolvimento, que para o contexto em que estávamos vivendo eram inadimissíveis.

Logo depois, o Laravel começou a utilizar muitos componentes do Symfony 2, um dos frameworks que mais admiro, e nesse ponto quando fui analisar novamente o framework me fiz a seguinte pergunta: Por que vou utilizar o Laravel, se posso usar diretamente o Symfony?

Por outro lado, e isso é fato, a popularidade do Laravel cresceu ainda mais e esse ponto não pode ser ignorado. Quando algo se torna popular, pode ter certeza que o mercado de trabalho será afetado instantaneamente.

Finalmente em 2015 tivemos o release da versão 5 do Laravel, e com muitas novidades em relação a versão 4. Também já podemos ver que é claro que o framework sim está aplicando muitos conceitos inegavelmente bem bacanas, principalmente para quem gosta de mágica em um framework.

Nesse ponto você deve estar se perguntando: Como assim mágica?

Sabemos que os frameworks tem o grande objetivo de agilizar nossa vida, evitando reuso de código e trazendo muitas funcionalidades prontas, por outro lado, existem frameworks que apesar de trazerem todos esses recursos e facilidades prontas, eles diretamente exigem que você trabalhe com todos esses componentes de forma explícita, ou seja, declarando tudo e como cada item vai ser utilizado. Um exemplo muito claro desse tipo de framework é o próprio Zend Framework 2. Cada Controller, Helper, Service, Factory, etc precisam estar claramente registrados, manualmente, nos arquivos de configuração.

Há desenvolvedores que não gostam desse tipo de abordagem, pois acham que o processo de desenvolvimento fica altamente burocratizado. Por outro lado, há desenvolvedores que preferem claramente isso, uma vez que eles tem certeza e controle de tudo que está acontecendo por trás do framework.

Um exemplo que gosto de mencionar em relação a isso é: Você pode ter um carro com câmbio manual e automático. Para muitos motoristas, usar câmbio manual é algo altamente mais prazeroso, pois eles podem ter mais "controle" do carro; por outro lado, há motoristas que simplesmente não querem ter 100% de controle, e não fazem a mínima questão de saber qual é a próxima marcha que ele utilizará.

Pois é. Eu poderia claramente dizer que o Zend Framework 2 é o carro com câmbio manual, já o Symfony, o com câmbio automático. Agora você deve estar se perguntando: e o Laravel?

O Laravel é carro da Google que dirige sozinho! Ele é tão mágico que realmente é a única comparação que encontrei. Agora veja bem, não estou dizendo que isso é bom ou ruim, estou dizendo que isso depende e muito do perfil do desenvolvedor.

Para que eu não fique dando apenas exemplos abstratos, vamos tentar pegar um exemplo concreto de um recurso do Laravel 5 chamado de: Method Injection.

Imagine que dentro de uma classe de Controller, você possui uma ação e essa ação necessita receber em uma dependência de uma classe qualquer.

Seguindo os exemplos que dei entre: ZF 2 e Laravel, temos as seguintes resoluções para esse "problema".

No Zend Framework 2:

Dentro da classe do controller, teríamos um método chamado de setMyObject(MyNamespaceMyObject $object).

```
1  // IndexController.php
2
3  <?php
4
5  namespace MeuModulo\Controller;
6
7  class IndexController extends AbstractActionController
8  {
9
```

```
10      private $myObject;
11
12      public function indexAction()
13      {
14          $myObject = $this->myObject;
15      }
16
17      public function setMyObject(\My\Namespace\MyObject $object)
18      {
19          $this->myObject = $object;
20      }
21  }
22
23  // IndexControllerFactory.php
24
25  <?php
26  namespace Market\Factory;
27
28  use Zend\ServiceManager\FactoryInterface;
29  use Zend\ServiceManager\ServiceLocatorInterface;
30
31  class IndexControllerFactory implements FactoryInterface
32  {
33      public function createService(ServiceLocatorInterface $controllerManager)
34      {
35          $allServices = $controllerManager->getServiceLocator();
36          $sm = $allServices->get('ServiceManager');
37
38          $object = $sm->get('MyObject');
39
40          $indexController = new \MeuModulo\Controller\IndexController();
41          $indexController->setMyObject($object);
42
43          return $indexController;
44      }
45  }
```

Perceba que no Zend Framework 2, criamos um método setter para poder ser executado e injetar nosso objeto em nosso controller.

Também temos uma factory que é chamada quando o controller é requisitado. Essa Factory instancia nosso controller, executa o método setMyObject injetando noss objeto no controller. Não quero que você entenda os detalhes sobre o Zend Framework nesse livro, porém, quero que você entenda

que a injeção de nosso objeto nesse controller foi algo totalmente explícito, ou seja, 0% mágica. Simplesmente, determinamos como um controller vai ser criado e quais objetos temos que injetar no mesmo.

Já no Laravel (um outro extremo), simplesmente precisamos disso para termos o mesmo efeito:

```
1 // IndexController.php
2 public function indexAction(\My\Namespace\MyObject $object)
3 {
4     $myObject = $object;
5 }
```

Apenas isso! No momento que você utiliza o Type Hint dizendo qual o tipo de objeto deve ser recebido nessa action, o próprio framework, magicamente, faz a injeção disso para você.

Entenda, isso não é errado, mas você precisa entender bem do framework para que você não perca o controle da sua aplicação.

Independente dos aspectos técnicos do Laravel, o mesmo ganhou explosivamente em popularidade de qualquer outro framework PHP do mercado (de acordo com a pesquisa do SitePoint[1]).

Isso jamais deve ser ignorado.

O Laravel sim, ainda possui suas inúmeras polêmicas implementações de design, que ao serem analisadas a fundo (como suas Façades, por exemplo), podem tirar do sério qualquer desenvolvedor mais experiente que realmente goste de trabalhar com design e padrão de projetos a risca. Além disso, se por um lado ele implementa em sua versão 5 recursos como: Events, Route Middleware, Contracts, Commands, etc; por outro lado, é muito claro que ainda ele não possui soluções tão elegantes para modularizar grandes aplicações como o Symfony e o Zend Framework o fazem com seus Bundles e Modules, gerando uma grande limitação no próprio framework.

O Laravel já é uma realidade no mundo do PHP e provavelmente ele não irá parar na versão 5.

Nesse livro, vou lhe apresentar de forma conceitual e prática como trabalhar com Laravel, utilizando seus principais recursos inseridos na versão 5, abusando de sua mágica, para tornar seu processo de desenvolvimento o mais rápido e simples possível.

[1] http://www.sitepoint.com/best-php-framework-2015-sitepoint-survey-results/

Instalação

Fazer a instalação do Laravel é algo extremamente simples. Basicamente, é como instalar qualquer outro framework PHP, porém, para isso, você deverá trabalhar com o Composer.

Não sei se alguma vez você já ouviu falar no composer. Ele é uma biblioteca escrita em PHP que tem o objetivo de fazer o gerenciamento de dependências de nossas aplicações.

Instalar o composer é uma tarefa extremamente simples:

- Acesse: http://getcomposer.org/download[2]
- Faça o download da biblioteca
- Pronto, você terá um arquivo chamado composer.phar em sua máquina pronto para uso.

Minha dica é de que você deixe esse arquivo registrado em suas variáveis de ambiente, para que ao digitar apenas o comando "composer" em seu terminal, ele já seja executado diretamente.

Uma vez que você possua o composer baixado / instalado em seu computador, instalar o Laravel é uma tarefa extremamente simples.

Instalação via Laravel Installer

O Laravel possui uma biblioteca que facilita muito sua vida para a criação de novos projetos Laravel, nós a chamamos de Laravel Installer.

Para instalá-la, você deverá utilizar o seguinte comando via composer:

```
1  composer global require "laravel/installer=~1.1"
```

Feito isso, não se esqueça de adicionar as bibliotecas do composer em suas variáveis de ambiente também. O caminho normalmente é:

```
1  ~/.composer/vendor/bin
```

Agora, basta simplesmente rodar o comando para criar um novo projeto Laravel:

[2] http://getcomposer.org/download

```
laravel new NomeDoProjeto
```

Instalação direta via composer

Uma outra forma de você instalar um novo projeto Laravel, é rodando o seguinte comando via composer:

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

Com isso o Laravel será interamente clonado e configurado em seu computador, estando pronto para uso.

Executando Laravel pela primeira vez

Uma vez que você executou um dos passos anteriores e se você possui no mínimo a versão 5.4 do PHP, basta entrar no diretório do projeto criado e executar o servidor embutido do PHP rodando o comando:

```
cd laravel
php -S localhost:8000 -t public/
```

Ou simplesmente digitar o comando abaixo:

```
php artisan serve
```

Prontinho! Laravel 5.1 rodando.

A estrutura do Laravel

Nesse capítulo, vou lhe apresentar a estrutura geral do funcionamento básico do framework, bem como sua estrutura de pastas e arquivos que são básicos necessários para que você consiga sair desenvolvendo.

Estrutura de pastas e arquivos

Quem um dia já chegou a instalar o Laravel 5, provavelmente deve ter tomado um ligeiro susto pela grande quantidade de arquivos e pastas que temos que entender para conseguir trabalhar com o framework.

De qualquer forma, para começar, você não precisa entender necessariamente todas as pastas e arquivos de configuração, nesse ponto, apresentarei apenas os que acho mais importantes nesse momento.

Estrutura de pastas

vendor

A pasta vendor, é gerada automaticamente pelo composer. Nessa pasta você encontrará todas as bibliotecas necessárias para rodar o framework. Você nunca deverá mexer nesse diretório, uma vez que ele é gerenciado automaticamente pelo próprio composer.

public

A pasta public, é uma das mais importantes da aplicação, uma vez que ela é o DocumentRoot do nosso sistema, ou seja, todas as requisições sempre cairão diretamente nela, dessa forma, isso garantirá que quem acessa nossa aplicação, jamais terá acesso a pastas do mesmo nível que a public.

Além disso, a pasta public é a responsável por servir, publicamente, todos os nossos assets para o "mundo exterior", ou seja, no final das contas, todos os javascripts, css, fonts, imagens, etc deverão estar na pasta public; sem contar que o arquivo index.php também ficará armazenado nela.

config

Pelo fato do Laravel ser um framework com diversos tipos de funcionalidades, o mesmo necessita de uma estrutura que nos permita fazer as mais diversas configurações de seus componentes. Nesse ponto, a pasta config é a responsável por armazenar os mais diversos arquivos de configuração do framework, desde configurações de banco de dados, envio de emails, cache, etc.

storage

Toda vez que você necessitar gravar qualquer arquivo dentro do Laravel, a pasta storage será muito útil. Ela tem o objetivo de armazenar os mais diversos arquivos para uso interno do framework, como arquivos de cache, banco de dados (sqlite por exemplo), entre outros.

databases

Na pasta databases você encontrará os recursos necessários para que você possa trabalhar com suas *migrations* e *seeders*. Veremos nos próximos capítulos maiores detalhes sobre esses recursos.

tests

O Laravel já vem totalmente configurado para que você consiga rodar seus testes automatizados. É na pasta tests que seus scripts de testes deverão ficar armazenados.

app

Essa é a pasta principal de nossa aplicação. Basicamente, grande parte do código que digitaremos em nosso projeto estará dentro dela. A pasta app, possui muitas subpastas, nesse ponto, pretendo explicá-las apenas quando tivermos a necessidade de fazer uso das mesmas.

Arquivos

composer.json e composer.lock

Esses dois arquivos são utilizados pelo composer para gerenciar todas as dependências do projeto: como biblioteca de terceiros e principalmente as do próprio Laravel.

.env e .env.example

O Laravel, trabalha com uma lib chamada de DotEnv. Ela tem o objetivo de guardar todas as informações de configuração de seu ambiente, como user, senha de banco de dados, informações de debug, entre outros. O arquivo .env nunca deverá fazer parte do seu controle de versão, uma vez que ele guarda informações específicas de um ambiente, logo, para fazer uso dele, basta copiar o arquivo .env.example para .env e fazer suas configurações locais.

artisan

O Artisan será o seu melhor amigo na jornada com o Laravel, ele lhe dará acesso por linha de comando para executar diversos tipos de tarefas, como: criar um Model, Controller, Providers, Commands, Migrations, etc. Com certeza você utilizará bastante esse camarada.

gulpfile.js

O gulpfile.js é o responsável por definir quais as tarefas que nosso *asset manager* chamado de Elixir executará.

package.json

O package.json é o arquivo responsável por definir as dependências de módulos Node.js que nossa aplicação possui. Por padrão, ele vem com duas dependências: a do Gulp e a do Laravel-elixir.

phpspec.yml

O phpspec.yml é o arquivo de configuração de um framework de testes que também é suportado pelo Laravel.

phpunit.xml

O phpunit.xml é o arquivo responsável por fazer todas as configurações básicas para que você também consiga rodar seus testes desenvolvidos utilizando o PHPUnit.

Ciclo de vida de uma aplicação Laravel

Quando acessamos uma aplicação Laravel, você deve imaginar que muita coisa deve acontecer por baixo dos panos do framework.

Explicarei brevemente, quais são os principais pontos que você deve conhecer quando fazemos uma requisição em uma aplicação.

Tudo passa pelo index.php

Quando fazemos uma requisição em uma aplicação Laravel, todas as URLs acessadas são apontadas diretamente para um único arquivo, o **index.php**, que fica localizado na pasta public.

Exemplo de um index.php padrão do Laravel 5:

```php
<?php

require __DIR__.'/../bootstrap/autoload.php';

$kernel = $app->make('Illuminate\Contracts\Http\Kernel');

$response = $kernel->handle(
        $request = Illuminate\Http\Request::capture()
);

$response->send();

$kernel->terminate($request, $response);
```

Como você pode perceber, o index.php do Laravel é muito simples, contudo, extremamente poderoso. É a partir dele que todo o ciclo da aplicação se inicia.

O primeiro passo, é dar um *require* no arquivo de autoload, garantindo que todos os arquivos necessários para a aplicação rodar sejam chamados, sem a necessidade de ficarmos dando *require*, manualmente, em todos os nossos arquivos.

Seguindo essa idéia, ele gera uma instância do Http Kernel, que é o componente responsável por processar uma imensa lista de *bootstrappers* (inicializadores) que terão a responsabilidade de capturar as variáveis de ambiente, rodar todas as configurações, handlers, logs, etc. Tudo isso é feito, antes mesmo da requisição ser processada.

Além disso, ele é o responsável por inicializar todos os *HTTP middlewares*, que nesse caso atuam como um plugin, mudando o comportamento da aplicação em tempo real. Um exemplo claro disso, é um middleware de sessão, que tem o objetivo de preparar toda a aplicação para trabalhar com sessions por exemplo.

Um dos pontos que não podemos deixar de citar, e que também é de responsabilidade do Kernel, é o de fazer as inicializações dos *Services Providers* de nossa aplicação. Os Services Providers são classes responsáveis por fazer o registro de todos os serviços no DiC (Dependency Injection Container) do Laravel.

Apesar do Kernel parecer algo extremamente complexo, podemos definir suas funções como extremamente simples:

- Fazer / inicializar os componentes do framework
- Receber um HTTP Request
- Retornar um Response

Se você perceber, o Kernel possui um método chamado de *handle*. Veja abaixo ele sendo utilizado:

```
$response = $kernel->handle(
        $request = Illuminate\Http\Request::capture()
);
```

O método handle, simplesmente recebe uma requisição (Request) e nos devolve uma Response, que será entregue. Apenas isso.

O processamento Interno de uma Request

Uma vez que o Kernel recebeu a requisição, e tudo já foi inicializado, o framework analisará a requisição para definir para qual recurso ela deverá encaminhá-la. Você deve estar pensando: "Mas como ele faz isso?"

O Laravel, não muito diferente de outros frameworks, trabalha com conceito de Rotas. Para tentar deixar a explicação mais didática, tentarei dar um exemplo fictício, vamos lá:

Exemplo

Vamos supor que nossa aplicação Laravel acaba de receber uma requisição com as seguintes informações:

```
GET /produtos HTTP/1.1
Host: www.meusite.com.br
```

Podemos perceber na requisição HTTP, simplificada acima, que a mesma foi enviada utilizando o método GET, solicitando informações de nosso recurso /produtos no host www.meusite.com.br.

Resumindo, alguém acessou o endereço: http://www.meusite.com.br/produtos

A funcionalidade de rotas do Laravel, simplesmente faz o mapeamento de qual método e recurso foi solicitado por uma requisição (nesse exemplo: Método GET e o recurso de produtos), e simplesmente aponta tal requisição para onde definirmos (pode ser uma função, controller, etc).

Segue um exemplo simples da utilização da funcionalidade de rotas do Laravel, processando a requisição que citamos como exemplo:

```
$app->get('produtos', function() {
        return "Meus Produtos";
});
```

Perceba que nesse caso, estamos dizendo que todas as requisições com o método GET que tentarem acessar o recurso /produtos, serão diretamente direcionadas para uma função anônima que retornará como response o valor: Meus Produtos.

Nos próximos capítulos desse livro, veremos mais a fundo como trabalhar com Rotas. O exemplo acima, foi apenas para demonstrar o funcionamento básico do ciclo de vida do framework.

Configurando sua aplicação Laravel

Alterando o Namespace de sua aplicação

No momento em que você navegar pelas pastas e arquivos criados pelo Laravel, você perceberá que o Namespace padrão gerado, automaticamente, pelo framework para você utilizar terá o nome de *App*.

O grande ponto, é que se em algum momento você resolver distribuir ou reaproveitar partes do código de sua app, você terá uma grande chance de sofrer colisões de nomes de suas classes, uma vez que o Namespace *App* é totalmente genérico e muito provavelmente você encontrará aplicações por aí utilizando tal nome.

Nesse ponto, o Laravel, nos provê um recurso, através do Artisan, para que possamos alterar o namespace padrão de nossa aplicação. É extremamente simples realizar essa tarefa. Veja abaixo:

```
1  php artisan app:name SeuNameSpace
```

Executando tal comando, o Laravel, fará automaticamente a mudança em todos os namespaces com o padrão *App* em sua aplicação para o novo namespace que você definiu.

Configurando banco dados

Por padrão, o Laravel te da a possibilidade de trabalhar com os seguintes sistemas de gerenciamento de banco de dados:

- Sqlite
- MySQL
- PostgreSQL
- SQL Server
- Redis *

Se você abrir o arquivo que está localizado em: *app/config/database.php*, você perceberá que já existe um array pré-definido com todas as configurações necessárias para que possamos ativar a utilização de quaisquer SGDBs.

Pontos para você ficar atento

Key “default”

Será nessa key que você definirá qual SGDB sua aplicação rodará, nesse ponto, você poderá escolher entre:

- sqlite
- mysql
- pgsql
- sqlsrv

Por outro lado, você verá que nos arrays de configuração de cada banco de dados, já temos um valor padrão, pré-definido, para user, password, host, etc; todavia, se você notar, perceberá que temos também uma função chamada: *env* os envolvendo.

Veja um exemplo abaixo:

```
'connections' => [

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

                'mysql' => [
                        'driver'    => 'mysql',
                        'host'      => env('DB_HOST', 'localhost'),
                        'database'  => env('DB_DATABASE', 'forge'),
                        'username'  => env('DB_USERNAME', 'forge'),
                        'password'  => env('DB_PASSWORD', ''),
                        'charset'   => 'utf8',
                        'collation' => 'utf8_unicode_ci',
                        'prefix'    => '',
                        'strict'    => false,
                ],

                'pgsql' => [
                        'driver'   => 'pgsql',
                        'host'     => env('DB_HOST', 'localhost'),
                        'database' => env('DB_DATABASE', 'forge'),
```

```
25                      'username' => env('DB_USERNAME', 'forge'),
26                      'password' => env('DB_PASSWORD', ''),
27                      'charset'  => 'utf8',
28                      'prefix'   => '',
29                      'schema'   => 'public',
30              ],
31
32              'sqlsrv' => [
33                      'driver'   => 'sqlsrv',
34                      'host'     => env('DB_HOST', 'localhost'),
35                      'database' => env('DB_DATABASE', 'forge'),
36                      'username' => env('DB_USERNAME', 'forge'),
37                      'password' => env('DB_PASSWORD', ''),
38                      'prefix'   => '',
39              ],
40
41      ],
42  ]
```

A função *env*, tem uma simples e ótima utilidade: Ela verifica se a key solicitada possui um valor em nosso arquivo *.env*, que fica na raiz do projeto. Caso a key exista, ela pegará o valor dessa key, caso contrário, ela utilizará o valor padrão que é passado em seu segundo parâmetro.

Nesse ponto, tudo que precisamos fazer, é definirmos as configurações de nossos dados sensíveis como: host, user, password, etc, em nosso arquivo *.env*, que automaticamente o Laravel, vai utilizá-las para estabelecer a conexão com o banco de dados.

Sempre lembrando que o arquivo .env, jamais deverá fazer parte de seu controle de versão.

Configurando envio de emails

Uma vez que você entendeu o conceito de como fazer as configurações de banco de dados no Laravel, o mesmo se aplica para as configurações de envio de email.

Basta abrir o arquivo *app/config/mail.php* e verificar quais keys de configuração estão sendo utilizadas por padrão pela a função *env*, nesse caso, basta colocar seus dados sensíveis em relação as configurações de email no arquivo *.env* de seu projeto.

Configuração do ambiente

Um item que não podemos deixar passar, sãos as configurações específicas de seu ambiente Laravel.

Veja um exemplo de pontos que você tem que se atentar e jamais deixar de configurar, corretamente, de acordo com o ambiente em que sua aplicação está rodando, ou seja: desenvolvimento, produção, etc.

```
1 APP_ENV=local
2 APP_DEBUG=true
3 APP_KEY=SomeRandomString
```

APP_ENV

Perceba que na Key APP_ENV, determinamos em qual ambiente a aplicação está sendo executada, ou seja, local, production, etc.

APP_DEBUG

Já a key APP_DEBUG é extremamente útil estar setada para true **apenas quando estamos em modo desenvolvimento**, pois dessa forma, todo erro que recebermos, poderemos ter todo seu tracing, facilitando a vida para entender o que realmente deu errado.

Nunca esqueça de deixar a key APP_DEBUG=false quando for colocar a aplicação em produção.

APP_KEY

A key APP_KEY é extremamente importante para definir sua aplicação Laravel como única, tendo uma chave para auxiliar nos processos de geração de tokens contra CSRF, Salt para o password hash, encriptação de cookies, sessions, etc.

Normalmente quando você instala o Laravel via composer, ele já preenche uma string randômica na sua APP_KEY, mas sempre é bom realmente verificar se esse ponto realmente foi executado com sucesso.

Concluindo

Ao acessar a pasta *app/config* do Laravel, você verá uma infinidade de arquivos que configuram funcionalidades específicas no framework.

O conceito sempre será o mesmo. Basta realizar as configurações das funcionalidades que realmente você terá a necessidade de utilizar.

No decorrer do livro, se precisarmos utilizar um recurso que ainda não configuramos, com certeza voltaremos nessa pasta *app/config*.

Trabalhando com Rotas, Controllers e Views

Introdução

Sei que até o momento, você teve que acompanhar muita teoria para entender:

- O Contexto atual do Laravel no mundo PHP
- Seu ciclo de vida
- Instalação
- Configurações iniciais

A partir desse capítulo, colocaremos um pouco mais a mão na massa para que você possa começar a sentir a grande flexibilidade e facilidade que o Laravel possui para desenvolver aplicações.

Voltando às Rotas

Quando nossa aplicação recebe uma requisição, o kernel do Laravel tem a função de inicializar seus principais componentes para que então possamos tratar o request.

Parte desse "tratamento" do request é realizado pelas rotas, todavia, temos "N" maneiras de trabalharmos e manipularmos tais rotas.

Nesse capítulo, apresentarei apenas duas formas de trabalharmos com rotas, para que possamos começar a ter as primeiras interações com nossos Controllers e Views.

Rotas e funções anônimas

Quando definimos uma rota, podemos fazer com que, uma vez que ela de um *match* com a request solicitada, ela execute uma função.

O arquivo de configuração de rotas do Laravel, pode ser encontrado em: *app/Http/routes.php.*

Exemplo:

```
//app/Http/routes.php

Router::get('produtos', function() {
    return "Olá mundo";
});
```

Perceba que nesse caso, quando algúem acessar o endereço: /produtos, via método GET, receberá um retorno instantâneo do Laravel, com o nosso famoso "Olá mundo", ou seja, nossa função anônima será imediatamente executada.

Por outro lado, muitas vezes, queremos encaminhar essa requisição para um camarada que possa fazer melhor a intermediação entre as camadas (responsabilidades) de nossa aplicação. Normalmente esse camarada é o nosso *Controller*.

*Ainda nesse capítulo, aprofundaremos um pouco mais sobre controllers. *

Por hora, entenda o Controller como uma classe que possui métodos; e no caso de nossas rotas, podemos fazer com que elas encaminhem nossa requisição exatamente para uma determinada classe (controller) e para um determinado método (action).

Vejamos um exemplo:

```
Router::get('produtos','ProdutosController@index');
```

No exemplo acima, estamos dizendo que quando recebermos uma request para a URI /produtos, com o método HTTP GET, executaremos o método *index* de nossa classe *ProdutosController*.

Controllers

Quando falamos de controllers, estamos falando de uma camada de nossa aplicação que possui uma responsabilidade muito bem definida: Ser o intermediador, ou seja, ele recebe a requisição, se houver necessidade ele pode chamar *models* para consultar ou mesmo persistir dados e depois definir como a *response* será retornada.

Nesse ponto, podemos por exemplo retornar um HTML simples, um json, ou mesmo, definir uma *View* (outra camada de nossa aplicação), que ficará responsável por renderizar o código HTML a ser exibido ao usuário final.

Nesse capítulo, vamos focar em apenas receber uma requisição em nosso controller e encaminhar o *response* de três maneiras.

1. HTML simples como retorno
2. Json
3. Apontamento para uma *View*

   Perceba que estamos partindo de que nossa Rota esteja nos apontando para: Produtos-Controller@index, ou seja, classe: ProdutosController, método: index.

Retornando um HTML Simples

```
<?php namespace App\Http\Controllers;

class ProdutosController extends Controller {

        public function index()
        {
                return "Olá Mundo";
        }
}
```

Por padrão o Laravel mantém seus controllers em: *app/Http/Controllers*

Retornando um JSON

O Laravel é esperto / mágico o suficiente em perceber que quando retornamos um objeto ou um array, em nossas actions, ele deva fazer a conversão dessas informações para JSON.

```
<?php namespace App\Http\Controllers;

class ProdutosController extends Controller {

        public function index()
        {
                $dados = ['ola'=>'Mundo'];
                return $dados;
        }
}
```

Nesse caso, nosso array chamado de “dados”, automaticamente, será convertido para JSON para ser enviado como response.

Apontando para uma view

Normalmente, essa é a opção mais utilizada quando você deseja entregar um HTML para o usuário final.

Para que você não precise ficar misturando o código HTML dentro de sua classe de controller, como vimos no exemplo "Retornando um HTML Simples", você pode apontar o retorno para a camada *View* do Laravel para que ela seja chamada e renderize o HTML para você.

Além disso, você contará com o recurso da *template engine* chamada de *Blade*, que provê uma série de facilidades, que aos poucos demostrarei para você.

Vamos ao código:

```
1  <?php namespace App\Http\Controllers;
2
3  class ProdutosController extends Controller {
4
5          public function index()
6          {
7                  $nome = 'Wesley";
8                  return view('produtos', ['nome'=>$nome]);
9          }
10 }
```

Perceba que no exemplo acima, estamos chamando um helper *view* passando dois parâmetros, onde o primeiro define qual será a *view* que será renderizada e o segundo, de quais dados atribuiremos para essa *view*.

Nesse caso, estamos definindo que a *view* **produtos** seja renderizada, e que a variável "$nome", poderá ser acessada de dentro dessa view.

Agora basta criarmos nossa view em: *resources/views/produtos.blade.php*.

```
1  <html>
2  <head>
3          <title>Produtos</title>
4  </head>
5
6  <body>
7
8  <h1>Olá <?php echo $this->nome; ?></h1>
9
10 </body>
11 <html>
```

Estamos utilizando basicamente código HTML. O código PHP tem seu uso apenas para imprimir valores que foram atribuídos em nossa view; dessa forma, não precisaremos ficar misturando toda nossas regras de negócios, entre outros, juntamente com nosso código HTML.

Agora, para simplificar ainda mais, podemos fazer uso do *Blade*, para não precisarmos utilizar sequer código PHP nessa view.

```
<html>
<head>
        <title>Produtos</title>
</head>

<body>

<h1>Olá {{ $nome }}</h1>

</body>
<html>
```

Nesse caso, estamos utilizando uma tag específica do Blade para imprimir o valor de uma varíavel: {{ $home }}.

Nos próximos capítulos, veremos como podemos fazer chamadas mais complexas dentro de nossos controllers e exibir informações em nossas views, utilizando o Blade, com mais detalhes.

Por hora, ficarei muito contente se você conseguiu entender esse pequeno fluxo que fizemos em uma aplicação Laravel, desde as suas rotas até a exibição de um conteúdo a partir de uma view.

Models

Introdução

Quando falamos do padrão M-V-C, estamos falando em separação de responsabilidades de diversos participantes de nossa aplicação.

No capítulo anterior, apresentei brevemente a utilização do "C" (Controller) e do "V" (View).

Nesse capítulo, focaremos em entender os conceitos básicos sobre o "M" (Model), e como poderemos fazer a utilização e **integração** dessa camada com as outras duas.

Falando sobre Models

Diferentemente do que muita gente pensa, o conceito de Models é bem abrangente. Quando falamos de Models, estamos falando de um participante que terá o objetivo de fazer qualquer consumo e transação de dados em nossa aplicação.

Vamos imaginar que desenvolveremos uma simples aplicação que fará a exibição de diversas informações sobre tweets.

Nesse caso, quem fará o consumo e gerenciamento das informações (dados) dos tweets para nossa aplicação, será exatamente nossa camada *Model*.

Entenda, quando falamos em Model, falamos em dados!

Quando falamos na camada de Model, muita gente já faz a associação direta de que são os Models que "cuidam" diretamente da parte de banco de dados de nossa aplicação.

Na realidade, a parte de banco de dados, é apenas uma, das mais diversas fontes de dados que nossa Model pode gerenciar, todavia, como é extremamente comum termos bancos de dados atrelados as nossas aplicações, a associação entre Models e Banco de dados fica, muito mais evidente do que outras fontes de dados.

De qualquer forma, sempre tenha em mente que quando estamos falando de Models, estamos falando de qualquer tipo de fonte de dados, como:

- Webservice / API
- Banco de dados
- Arquivos textos
- entre outros

Models no Laravel

Normalmente, quando estamos falando de Models no Laravel, estamos falando de como faremos acesso as informações de banco de dados em nossa aplicação, bem como faremos a disponibilização e transações com tais informações.

O Laravel possui um ORM chamado de Eloquent, o qual faz a relação direta entre as nossas classes de Model com nossas tabelas de banco de dados.

Acredito que o grande *goal* do Eloquent, é a sua excepcional facilidade de uso, nesse ponto, mostrarei brevemente como você poderá fazer uso dessa poderosa ferramenta.

Configurando um banco de dados

Como quase tudo no Laravel, fazer a configuração de um banco de dados é algo extremamente simples. Basicamente, você terá de definir qual será o SGDB a ser utilizado e passar as informações de acesso e/ou credenciais.

O arquivo responsável por conter todos os SGDBs pré-configurados no Laravel é:

config/database.php

Ele é composto por um array, onde facilmente poderá ser modificado.

Segue um exemplo de uma pequena parte desse arquivo:

```
1  return [
2
3          /*
4          |--------------------------------------------------------------------------
5          | Default Database Connection Name
6          |--------------------------------------------------------------------------
7          |
8          | Here you may specify which of the database connections below you wish
9          | to use as your default connection for all database work. Of course
10         | you may use many connections at once using the Database library.
11         |
12         */
13
14         'default' => 'mysql',
15
16         /*
17         |--------------------------------------------------------------------------
18         | Database Connections
19         |--------------------------------------------------------------------------
```

```
20          |
21          | Here are each of the database connections setup for your application.
22          | Of course, examples of configuring each database platform that is
23          | supported by Laravel is shown below to make development simple.
24          |
25          |
26          | All database work in Laravel is done through the PHP PDO facilities
27          | so make sure you have the driver for your particular database of
28          | choice installed on your machine before you begin development.
29          |
30          */
31
32          'connections' => [
33
34                  'sqlite' => [
35                          'driver'   => 'sqlite',
36                          'database' => storage_path().'/database.sqlite',
37                          'prefix'   => '',
38                  ],
39
40                  'mysql' => [
41                          'driver'    => 'mysql',
42                          'host'      => env('DB_HOST', 'localhost'),
43                          'database'  => env('DB_DATABASE', 'forge'),
44                          'username'  => env('DB_USERNAME', 'forge'),
45                          'password'  => env('DB_PASSWORD', ''),
46                          'charset'   => 'utf8',
47                          'collation' => 'utf8_unicode_ci',
48                          'prefix'    => '',
49                          'strict'    => false,
50                  ],
51  ]
```

Perceba que temos um *key* nesse array chamada de default. Essa *key* é a responsável por definir qual será o banco de dados padrão a ser utilizado em nossa aplicação.

Como você pode perceber, o padrão trazido pelo Laravel é o MySQL.

O outro ponto que você deve notar, é que temos uma *key* chamada MySQL, onde podemos informar todas as configurações de acesso ao banco de dados, contudo, se você perceber, o valor das principais informações estão setados utilizando uma função chamada *env*, por exemplo:

```
'host' => env('DB_HOST', 'localhost'),
```

Isso significa que o Laravel por padrão acessará nosso arquivo **.env** e buscará pela *key* DB_HOST, caso ele não a encontre, o valor padrão para essa informação será "localhost".

Lembre-se, o arquivo .env, tem o objetivo de armazenar as informações sensíveis de ambiente de nossa aplicação, nesse ponto, podemos ter algo desse tipo ao abrirmos esse arquivo.

```
DB_HOST=localhost
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
```

Será exatamente nesse arquivo que você entrará com as informações de acesso ao seu banco de dados.

Utilizando o sqlite como exemplo

Apenas para fins didáticos, utilizaremos um banco sqlite para trabalhar com os exemplos desse livro, logo, realizarei apenas dois pequenos passos para ter meu banco de dados configurado.

1. Mudar a chave default do meu arquivo database.php para: sqlite
2. Criar um arquivo em banco chamado database.sqlite dentro da pasta storage da aplicação

Criando nosso primeiro Model

Criar um model no Laravel é algo extremamente simples, basta criar uma classe que estenda de: *IlluminateDatabaseEloquentModel*, por outro lado, o framework nos traz diversas facilidades para que criemos nossos models, utilizando a linha de comando com o *Artisan*.

Basta rodar o comando:

```
php artisan make:model Produto -m
```

E você terá o seguinte resultado:

```
Model created successfully.
Created Migration: 2015_04_20_213916_create_produtos_table
```

O Artisan, acaba de criar dois arquivos:

- Produto.php, dentro de nossa pasta app
- 2015_04_20_213916_create_produtos_table.php, dentro da pasta database/migrations

O parâmetro *-m*, foi utilizado para que o Laravel crie além do model, uma migration correspondente.

Falando brevemente sobre Migrations

O Laravel possui o recurso de *migrations*, onde tem o objetivo de gerenciar cada mudança estrutural de nosso banco de dados, ou seja, para cada tabela, coluna, índice, criados em nosso banco de dados, podemos ter uma migration para realizar essa operação de forma automática.

Um ponto bem interessante sobre *migrations*, é que temos exatamente cada versão da estrutura de nossa banco de dados, logo, se sentirmos a necessidade, podemos facilmente dar um "roll back".

Seguinte nosso exemplo do model criado: Produto, a migration 2015_04_20_213916_create_produtos_table foi criada:

```
1  <?php
2
3  use Illuminate\Database\Schema\Blueprint;
4  use Illuminate\Database\Migrations\Migration;
5
6  class CreateProdutosTable extends Migration {
7
8          /**
9           * Run the migrations.
10          *
11          * @return void
12          */
13         public function up()
14         {
15                 Schema::create('produtos', function(Blueprint $table)
16                 {
17                         $table->increments('id');
18                         $table->timestamps();
19                 });
20         }
21
22         /**
23          * Reverse the migrations.
```

```
24          *
25          * @return void
26          */
27         public function down()
28         {
29                 Schema::drop('produtos');
30         }
31
32 }
```

Se você perceber, temos dois métodos principais: up() e down(). O método **up** é executado quando a migration é rodada, já o método **down**, é executado cada vez que damos um roll back, nessa mesma migration.

Nesse nosso caso, perceba que a o método **up** está sendo responsável por uma tabela chamada de *produtos*.

Por padrão, tal tabela possuirá uma coluna: ID, auto_increment e outras duas colunas referentes a data / hora de inserção e alteração de um registro (created_at e updated_at).

Para que possamos seguir com nosso exemplo, adicionarei mais duas colunas em nossa tabela produtos:

1. nome, com tamanho 255
2. descricao, do tipo text

```
1  public function up()
2  {
3      Schema::create('produtos', function(Blueprint $table)
4      {
5          $table->increments('id');
6          $table->string('nome', 255);
7          $table->text('descricao');
8          $table->timestamps();
9      });
10 }
```

Feita essa nossa alteração, basta apenas rodarmos os seguinte comandos via Artisan:

```
1  php artisan migrate
```

O comando migrate:install, cria uma tabela em nosso banco de dados para poder gerenciar nossas migrations.

Já o comando migrate, executa todas as migrations disponíveis em nossa aplicação.

Pronto! Já temos uma tabela de banco de dados criada, pronta para ser utilizada em conjunto com nosso Model Produto.

Brincando com nosso Model

Se você abrir o arquivo de nosso Model, localizado em: appProduto.php, você terá o seguinte resultado:

```
1  <?php namespace App;
2
3  use Illuminate\Database\Eloquent\Model;
4
5  class Produto extends Model {
6
7          //
8
9  }
```

Apesar de ser um arquivo extremamente pequeno, ele estende de uma classe chamada Model, que possui uma infinidade de recursos para você manipular sua tabela de banco de dados.

Vamos brincar um pouco com nosso Model, utilizando o **tinker**, um console interativo, disponibilizado pelo próprio Laravel.

```
1  php artisan tinker
2  Psy Shell v0.4.4 (PHP 5.6.4 — cli) by Justin Hileman
3  >>> use App\Produto;
4  => false
5  >>> $produto = new Produto();
6  => <App\Produto #0000000002ad4a98000000014d4e26e2> {}
7  >>> $produto->nome = "Livro de Laravel 5";
8  => "Livro de Laravel 5"
9  >>> $produto->descricao = "Descricao do livro";
10 => "Descricao do livro"
11 >>> $produto->save();
12 => true
```

Perceba que ao acessarmos o *tinker*, criamos um objeto Produto (baseado em nosso Model), e setamos dois atributos: nome e descricao; depois disso, apenas executamos o método save() e pronto. Já temos nosso primeiro registro gravado em nosso banco de dados!

Realmente é MUITO simples trabalhar com Models baseados no *Eloquent*.

Vamos um pouco além.

```
>>> Produto::all();
=> <Illuminate\Database\Eloquent\Collection #00000005e90814000000001112d4826> [
       <App\Produto #00000005e90814300000001112d4826> {
           id: "1",
           nome: "Livro de Laravel 5",
           descricao: "Descricao do livro",
           created_at: "2015-04-20 22:01:57",
           updated_at: "2015-04-20 22:01:57"
       }
   ]
```

Ao chamarmos estaticamente o método **all()** de nosso model, perceba que recebemos uma coleção de modelos Produto (Eloquent), que nesse caso, é o produto que acabamos de inserir no banco de dados.

Se quisermos, podemos selecionar apenas o Produto desejado, utilizando o método **find()**.

```
>>> $livro = Produto::find(1);
=> <App\Produto #00000005e90814600000001112d4826> {
       id: "1",
       nome: "Livro de Laravel 5",
       descricao: "Descricao do livro",
       created_at: "2015-04-20 22:01:57",
       updated_at: "2015-04-20 22:01:57"
   }
```

Nesse caso, o Model Produto, com o ID 1, foi atribuído em nossa variável $livro.

Podemos agora, fazer modificações nesse objeto e persisti-las novamente no banco de dados:

```
>>> $livro->descricao = "O melhor livro de Laravel, ever!";
=> "O melhor livro de Laravel, ever!"
>>> $livro->save();
=> true
```

Protinho, mudamos facilmente a descrição de nosso produto.

Vamos listar novamente os produtos, para ver se nossa modificação foi refletida:

```
>>> Produto::all();
=> <Illuminate\Database\Eloquent\Collection #000000005e90814300000001112d4826> [
       <App\Produto #000000005e90814100000001112d4826> {
           id: "1",
           nome: "Livro de Laravel 5",
           descricao: "O melhor livro de Laravel, ever!",
           created_at: "2015-04-20 22:01:57",
           updated_at: "2015-04-20 22:10:48"
       }
   ]
```

Pronto, podemos ver que os dados foram alterados no banco de forma totalmente descomplicada.

Utilizando nosso Model no Controller

Agora que já demos nossos primeiros passos com nosso Model, vamos exibir tais informações no browser do usuário, seguindo exatamente o fluxo que vimos no capítulo anterior, todavia, aplicando agora nossos conhecimentos sobre Models.

```
// app/Http/Controllers/ProdutosController.php
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Produto;

class ProdutosController extends Controller
{
    public function index()
    {
        $produtos = Produto::all();

        return view('produtos',['produtos'=>$produtos]);
    }
}
```

Perceba que estamos atribuindo para a variável produtos, uma coleção de produtos trazida pelo nosso Model, depois disso, apenas atribuimos tal coleção para nossa View (produtos.blade.php).

```
// resources/views/produtos.blade.php

<html>
<head>
    <title>Produtos</title>
</head>
<body>

<h1>Produtos</h1>

<ul>
    <?php foreach($produtos as $produto): ?>
    <li><?php echo $produto->nome; ?> (<?php echo $produto->descricao;?>)</li>
    <?php endforeach; ?>
</ul>

</body>
<html>
```

Com isso teremos nosso resultado tão desejado, uma listagem dos dados do banco de dados em nossa view:

Listagem de Produtos

Agora perceba que utilizamos as tags do PHP para fazermos o foreach; para facilitar nossa vida, podemos utilizar o recurso de foreach do próprio Blade para facilitar.

```
<html>
<head>
    <title>Produtos</title>
</head>
<body>

<h1>Produtos</h1>

<ul>
    @foreach($produtos as $produto)
        <li>{{ $produto->nome }} ({{ $produto->descricao }})</li>
    @endforeach
</ul>

</body>
<html>
```

Perceba que tivemos o mesmo resultado, todavia com um código muito mais limpo.

Rumo ao primeiro CRUD

Normalmente, quem já possui uma certa experiência com desenvolvimento web, provavelmente já está familiarizado com o termo *CRUD*, que significa: Create, Retrieve, Update e Delete.

O *CRUD* nada mais é do que um termo utilizado representando as principais operações que podemos realizar em uma entidade, ou seja, Criar, Listar, Alterar e Remover dados.

Listando nossos Produtos

No capítulo anterior, quando entendemos os conceitos iniciais sobre Models, fizemos uma simples listagem de nossos produtos. Agora, iremos preparar tal listagem para que possamos executar nossas principais operações (CRUD).

Herdando um template

Acredito que você já deva ter se antecipado e feito a seguinte pergunta: Como podemos aproveitar um template (view) padrão para todas as nossas páginas, sem que precisemos ficar duplicando a todo momento nosso HTML que nos serve como base de nossa aplicação?

A resposta para essa pergunta é: **Blade**.

Essa é uma das grandes vantagens de você poder utilizar o Blade. O Blade possibilita que você herde suas views de um template principal a fim de facilitar o reaproveitamento de código.

Vamos ver como isso funciona na prática.

Hoje, nossa view: produtos.blade.php encontra-se da seguinte forma:

```
1  <html>
2  <head>
3      <title>Produtos</title>
4  </head>
5  <body>
6
7  <h1>Produtos</h1>
8
9  <ul>
10     @foreach($produtos as $produto)
11         <li>{{ $produto->nome }} ({{ $produto->descricao }})</li>
```

```
12      @endforeach
13  </ul>
14
15  </body>
16  <html>
```

Nossa idéia, é utilizarmos um template padrão que o Laravel já nos trás para facilitar nosso trabalho.

Apenas lembrando que você tem toda a possibilidade de customizar e criar seu próprio template padrão para sua aplicação.

Na versão 5.1 do Laravel, você terá de criar seu template base para ser utilizado por toda a aplicação - na versão 5 do framework, por padrão tínhamos um arquivo chamado **app.blade.php** dentro da pasta *resources/views*.

Para facilitar sua sequência na leitura, colocarei aqui o arquivo **app.blade.php**:

```
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4           <meta charset="utf-8">
5           <meta http-equiv="X-UA-Compatible" content="IE=edge">
6           <meta name="viewport" content="width=device-width, initial-scale=1">
7           <title>Laravel</title>
8
9           <link href="{{ asset('/css/app.css') }}" rel="stylesheet">
10
11          <!-- Fonts -->
12          <link href='//fonts.googleapis.com/css?family=Roboto:400,300' rel='stylesheet' \
13  type='text/css'>
14
15          <!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media quer\
16  ies -->
17          <!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
18          <!--[if lt IE 9]>
19                  <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
20                  <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
21          <![endif]-->
22  </head>
23  <body>
24          <nav class="navbar navbar-default">
25                  <div class="container-fluid">
```

```
26                          <div class="navbar-header">
27                                  <button type="button" class="navbar-toggle collapsed" data-toggle=
28  data-target="#bs-example-navbar-collapse-1">
29                                          <span class="sr-only">Toggle Navigation</span>
30                                          <span class="icon-bar"></span>
31                                          <span class="icon-bar"></span>
32                                          <span class="icon-bar"></span>
33                                  </button>
34                                  <a class="navbar-brand" href="#">Laravel</a>
35                          </div>
36
37                          <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
38                                  <ul class="nav navbar-nav">
39                                          <li><a href="{{ url('/') }}">Home</a></li>
40                                  </ul>
41
42                                  <ul class="nav navbar-nav navbar-right">
43                                          @if (Auth::guest())
44                                                  <li><a href="{{ url('/auth/login') }}">Login</a></
45                                                  <li><a href="{{ url('/auth/register') }}">Register
46                                          @else
47                                                  <li class="dropdown">
48                                                          <a href="#" class="dropdown-toggle" data-t
49  aria-expanded="false">{{ Auth::user()->name }} <span class="caret"></span></a>
50                                                          <ul class="dropdown-menu" role="menu">
51                                                                  <li><a href="{{ url('/auth/logout'
52                                                          </ul>
53                                                  </li>
54                                          @endif
55                                  </ul>
56                          </div>
57                  </div>
58          </nav>
59
60          @yield('content')
61
62          <!-- Scripts -->
63          <script src="//cdnjs.cloudflare.com/ajax/libs/jquery/2.1.3/jquery.min.js"></scr\
64  ipt>
65          <script src="//cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.3.1/js/bootst\
66  rap.min.js"></script>
67  </body>
```

```
68  </html>
```

Se você perceber, quase no final desse arquivo, você encontrará a seguinte chamada: **@yield('content')**. Isso significa que se criarmos em nossa view, uma **@section** com o nome de *content*, e herdarmos o template app.blade.php, todo o conteúdo de nossa view aparecerá exatamente no local onde temos a chamada: **@yield('content')**.

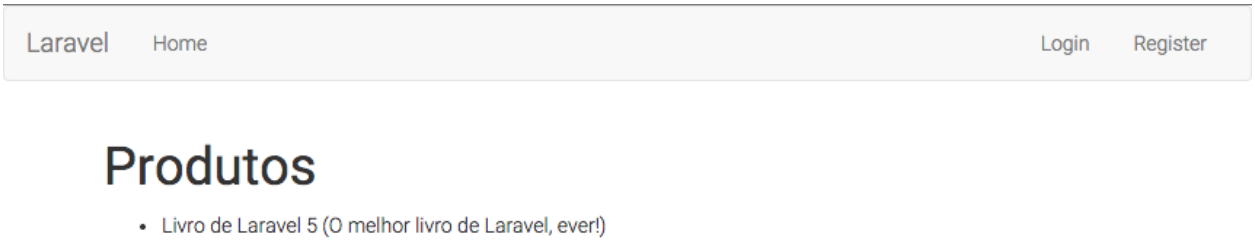
Para facilitar o entendimento, façamos isso na prática:

```
1   // produtos.blade.php
2   @extends('app')
3
4   @section('content')
5   <div class="container">
6       <h1>Produtos</h1>
7           <ul>
8               @foreach($produtos as $produto)
9               <li>{{ $produto->nome }} ({{ $produto->descricao }})</li>
10              @endforeach
11          </ul>
12  </div>
13  @endsection
```

Perceba que utilizamos o comando **@extends** do Blade para fazer com que nossa view produtos.blade.php possa herdar o template app.blade.php.

Em seguida, utilizamos a chamada **@section('content')** para definir que tudo que estiver dentro dela, será aplicado exatamente aonde temos a chamada **@yield('content')** no template app.blade.php.

Veja o resultado abaixo:



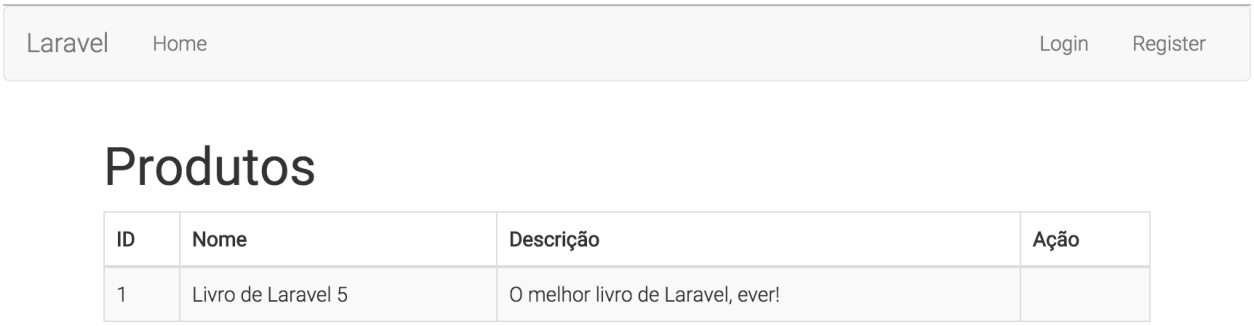
Listagem de produtos herdando template app.blade.php

Fazendo a listagem em uma tabela

Agora que já temos um template base para toda nossa aplicação, faremos a listagem dos produtos em uma tabela afim de facilitar a visualização, bem como adicionar as ações que poderemos realizar em cada um dos registros.

```
1  @extends('app')
2
3  @section('content')
4      <div class="container">
5          <h1>Produtos</h1>
6
7          <table class="table table-striped table-bordered table-hover">
8              <thead>
9              <tr>
10                 <th>ID</th>
11                 <th>Nome</th>
12                 <th>Descrição</th>
13                 <th>Ação</th>
14             </tr>
15             </thead>
16             <tbody>
17
18             @foreach($produtos as $produto)
```

```
19              <tr>
20                  <td>{{ $produto->id }}</td>
21                  <td>{{ $produto->nome }}</td>
22                  <td>{{ $produto->descricao }}</td>
23                  <td></td>
24              </tr>
25              @endforeach
26
27              </tbody>
28          </table>
29      </div>
30  @endsection
```



Listagem de produtos em uma tabela

Criando novos produtos

Agora que já temos nossa página de listagem de produtos devidamente formatada e utilizando um template base, iniciaremos o processo para criar novos produtos no sistema.

Criando página de “Create”

Nosso principal ponto nesse momento é criar uma action que chamaremos de *create* para que possamos apontar a seguinte rota para ela: produtos/create.

```php
// app/Http/Controllers/ProdutosController.php

<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Produto;

class ProdutosController extends Controller
{
    public function index()
    {
        $produtos = Produto::all();

        return view('produtos.index',['produtos'=>$produtos]);
    }

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

Perceba que além de termos criado o método create, também alteramos o retorno da view no método index de: view(‘produtos’) para view(‘produtos.index’).

Para facilitar a organização de nosso código, criaremos uma pasta chamada produtos dentro da pasta: resources/views, dessa forma, saberemos que todas as views referentes ao nosso CRUD de produtos estarão armazenados nessa pasta.

Também renomearemos o arquivo: resources/views/produtos.blade.php para resources/views/produtos/index.blade.php

Nesse ponto, vamos criar o arquivo: create.blade.php:

```
// resources/views/produtos/create.blade.php
@extends('app')

@section('content')
    <div class="container">
        <h1>Novo Produto</h1>

    </div>
@endsection
```

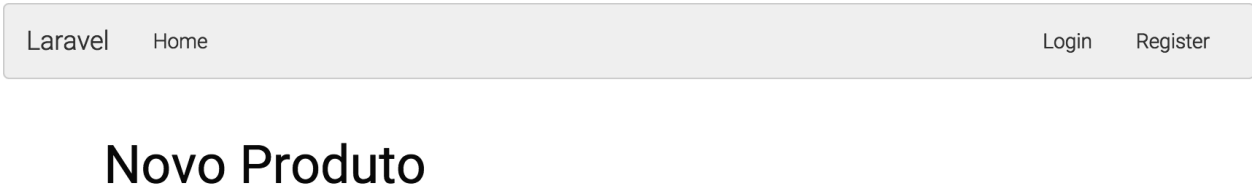
Falta registrarmos uma nova rota apontando: produtos/create para essa nossa nova action.

No arquivo routes.php faremos o seguinte registro:

```
//app/Http/routes.php

// ...
Route::get('produtos/create','ProdutosController@create');
// ...
```

Ao acessarmos a aplicação com a rota: produtos/create, teremos o seguinte resultado:



Listagem de produtos em uma tabela

Trabalhando com formulários no Laravel

Instalando o Illuminate/Html

Já temos nossa página produtos/create pronta e preparada para receber nosso formulário.

Você possui duas opções nesse momento:

1. Criar manualmente seus formulários HTML
2. Utilizar a ajuda do componente Illuminate/Html do Laravel.

Nesse nosso caso, utilizaremos a segunda opção para termos mais agilidade no processo de criação dos formulários.

A idéia é instalarmos o pacote: Illuminate/Html[3]

Para instalá-lo, basta o rodar o seguinte comando via composer. Você terá o resultado similar ao exibido logo abaixo:

```
composer require illuminate/html
Using version ~5.0 for illuminate/html
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing illuminate/html (v5.0.0)
    Loading from cache

Writing lock file
Generating autoload files
Generating optimized class loader
```

Teremos que registrar esse pacote como um "Service Provider" (falaremos mais sobre Service Providers nos próximos capítulos), dessa forma o Laravel poderá encontrar esses componentes.

Para fazer isso, adicione a seguinte linha ao final do array *providers* dentro do arquivo: **config/app.php**.

```
// config/app.php

providers' => [
// Outros providers
'Illuminate\Html\HtmlServiceProvider'
```

Também, no mesmo arquivo, você terá que adicionar duas linhas no array de *aliases*:

[3] https://github.com/illuminate/html

```
// config/app.php
'aliases' => [
// Outros aliases
'Form' => 'Illuminate\Html\FormFacade',
'Html' => 'Illuminate\Html\HtmlFacade',
```

Pronto! Agora, poderemos criar nossos formulários de uma forma muito rápida e produtiva.

Criando formulário

Uma vez que já temos o pacote IlluminateHtml instalado, podemos abrir nosso arquivo create.blade.php e fazermos um rápido teste:

```
//resources/views/protudos/create.blade.php
@extends('app')

@section('content')
    <div class="container">
        <h1>Novo Produto</h1>

        {!! Form::open() !!}

        {!! Form::close() !!}

    </div>
@endsection
```

Ao acessarmos nossa aplicação na rota produtos/create, aparentemente nada terá mudado, porém, quando inspecionarmos o código fonte, poderemos perceber que os seguintes itens foram adicionados:

```
<form method="POST" action="http://localhost:8000/produtos/create" accept-charse\
t="UTF-8">
<input name="_token" type="hidden" value="ZTbXf6wfDFxQ7f1gOWBz07SsAuoQv8ji90uUTu\
gz">
</form>
```

O Laravel, automaticamente, criou as tags <form> e ainda um elemento hidden como um token para proteger nosso formulário contra CSRF.

Vamos agora adicionar os outros itens para nosso formulário ficar completo:

```
@extends('app')

@section('content')
    <div class="container">
        <h1>Novo Produto</h1>

        {!! Form::open() !!}

        <!-- Nome Form Input -->

        <div class="form-group">
            {!! Form::label('nome', 'Nome:') !!}
            {!! Form::text('nome', null, ['class'=>'form-control']) !!}
        </div>

        <!-- Descricao Form Input -->

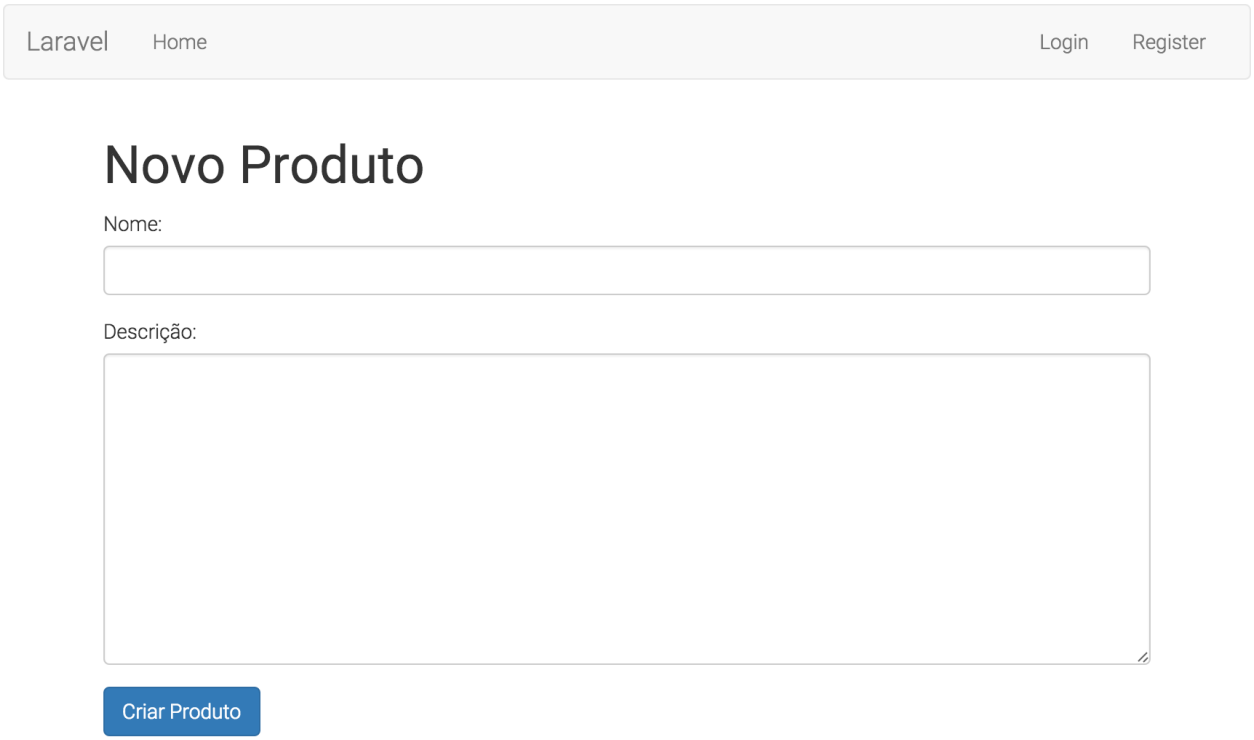
        <div class="form-group">
            {!! Form::label('descricao', 'Descrição:') !!}
            {!! Form::textarea('descricao', null, ['class'=>'form-control']) !!}
        </div>

        <div class="form-group">
            {!! Form::submit('Criar Produto', ['class'=>'btn btn-primary']) !!}
        </div>

        {!! Form::close() !!}

    </div>
@endsection
```

Perceba que com a simples utilização das tags mostradas acima, conseguimos o seguinte resultado em nosso formulário:



Formulário de inclusão de produto

A action “store”

Agora que já temos nosso formulário criado, precisamos criar uma nova action que será responsável por receber a request do nosso form e chamar nosso Model Produto para persistir os dados no banco. Também precisaremos registrar uma nova rota: produto/store.

Vamos lá:

```
1  // app/Http/routes.php
2
3  Route::post('produtos/store','ProdutosController@store');
```

Perceba que estamos utilizando **Route::post**, uma vez que o formulário será enviado via o método HTTP POST e não GET.

Também temos que realizar um pequeno ajuste em nosso arquivo create.blade.php para apontar o form para a nossa url produtos/store.

```
{!! Form::open(['url'=>'produtos/store']) !!}
```

Agora, vamos criar nossa action store:

```
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Produto;
use Illuminate\Http\Request;

class ProdutosController extends Controller
{
    public function index()
    {
        $produtos = Produto::all();

        return view('produtos.index',['produtos'=>$produtos]);
    }

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

    public function store(Request $request)
    {
        $input = $request->all();
        Produto::create($input);

        return redirect('produtos');

    }
}
```

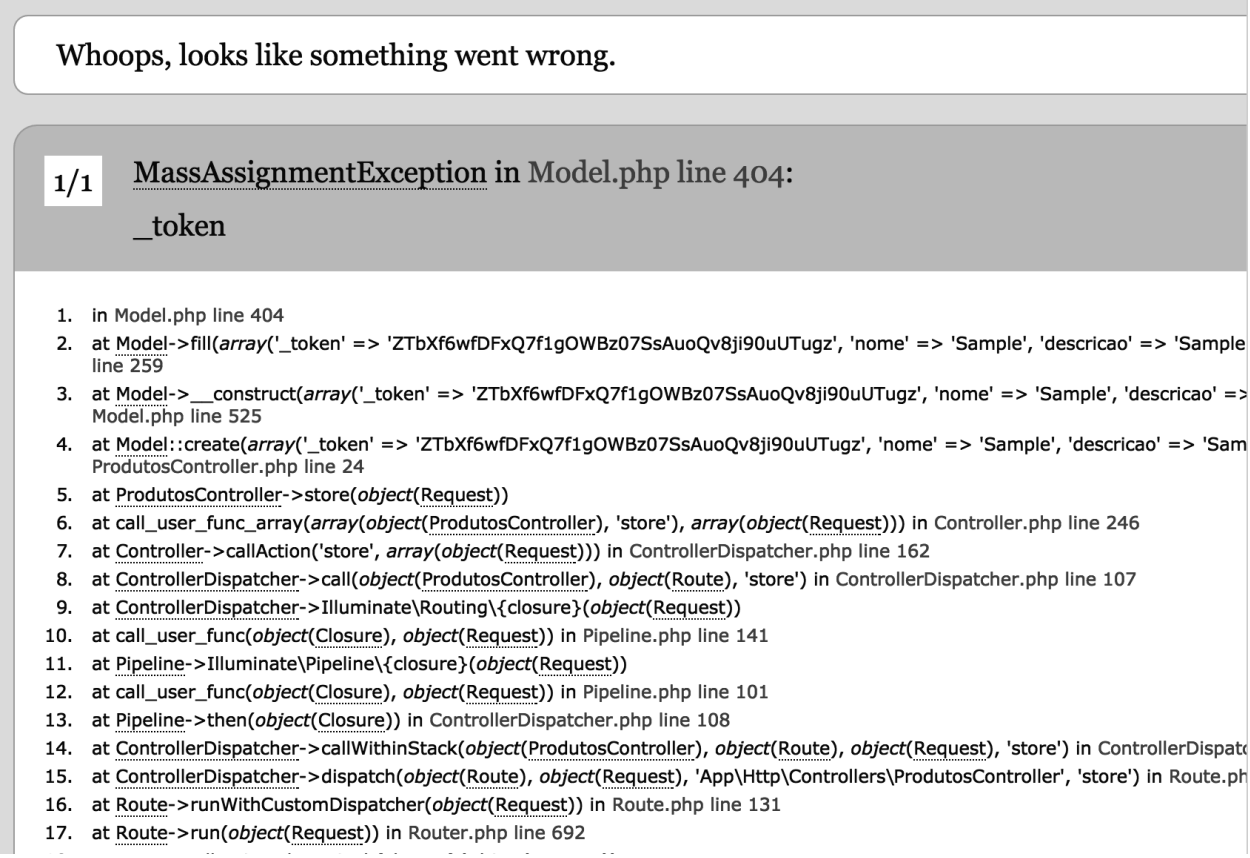
Perceba que em nossa action *store*, estamos recebendo por parâmetro um objeto request.

Armazenamos todas as informações enviadas via form de nosso request na variável $input.

Depois disso, chamamos o método create de nosso Model Produto, para inserir os dados no banco de dados.

Em seguida, redirecionamos o usuário para a listagem dos produtos novamente.

Mas a vida é uma caixinha de surpresas! Se você me acompanhou até aqui, deve ter topado com o seguinte erro ao submeter o formulário:



Erro ao submeter o formulário

O Laravel trabalha com uma pequena proteção em seus models contra *Mass Assigment*, ou seja, quando atribuimos dados em massa, como no caso de termos passado todas as informações do form diretamente.

Nesse caso, para permitir que possamos inserir os dados do form diretamente, temos que configurar nosso Model para aceitar a atribuição em massa de nossos dois campos: nome, descricao. Para isso, basta adicionar um atributo chamado $fillable na classe de nosso model Produto.

```
1  // app/Produto.php
2  <?php namespace App;
3
4  use Illuminate\Database\Eloquent\Model;
5
6  class Produto extends Model
7  {
8          protected $fillable = ['nome','descricao'];
9  }
```

Prontinho. Agora, quando você enviar os dados do form, o Laravel permitirá que os atributos nome e descricao sejam atribuidos diretamente para criarmos um registro.

Validando dados

Agora que já temos nosso formulário possibilitando a criação de novos registros, temos que poder validar as informações que estão sendo enviadas através das requisições.

O Laravel 5 possui um recurso que podemos chamar de **Requests personalizadas**, ou seja, criamos uma classe de Request e nela informamos os itens que devem ser validados.

Utilizaremos o artisan para gerar automaticamente essa classe:

```
1 php artisan make:request ProdutoRequest
2 Request created successfully.
```

O arquivo ProdutoRequest.php foi criado na pasta app/Http/Requests:

```
1  // app/Http/Requests/ProdutoRequest.php
2
3  <?php namespace App\Http\Requests;
4
5  use App\Http\Requests\Request;
6
7  class ProdutoRequest extends Request
8  {
9
10     /**
11      * Determine if the user is authorized to make this request.
12      *
13      * @return bool
14      */
15     public function authorize()
16     {
17         return false;
18     }
19
20     /**
21      * Get the validation rules that apply to the request.
22      *
23      * @return array
24      */
25     public function rules()
26     {
27         return [
28             //
```

```
29          ];
30      }
31
32  }
```

Perceba que temos dois métodos nessa classe: authorize() e rules().

O método *authorize()* determina se temos autorização de realizar essa request, logo, se ele retornar true, quer dizer que ela pode ser executada. Entenda que você poderá colocar a regra que quiser para permitir ou não a requisição de determinado usuário.

Já o método *rules()*, é responsável por validar cada um dos dados enviados na request.

O Laravel 5 já possui uma série de validadores padrão que podem ser encontrados em sua Documentação Oficial[4].

Nesse caso, faremos as seguintes alterações em nossa classe ProdutoRequest:

```
1   <?php namespace App\Http\Requests;
2
3   use App\Http\Requests\Request;
4
5   class ProdutoRequest extends Request
6   {
7
8       /**
9        * Determine if the user is authorized to make this request.
10       *
11       * @return bool
12       */
13      public function authorize()
14      {
15          return true;
16      }
17
18      /**
19       * Get the validation rules that apply to the request.
20       *
21       * @return array
22       */
23      public function rules()
24      {
25          return [
```

[4] http://laravel.com/docs/5.0/validation#available-validation-rules

```
            'nome' => 'required|min:5',
            'descricao' => 'required'
        ];
    }
}
```

Perceba que estamos agora retornando *true* no método authorize(), e definimos que no método rules() os atributos nome e descricao são obrigatórios e que o atributo nome também precisa conter no mínimo 5 caracteres.

Agora basta injetarmos essa classe como parâmetro em nossa action *store*:

```
// app/Http/Controllers/ProdutosController.php

<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Produto;
use App\Http\Requests\ProdutoRequest;

class ProdutosController extends Controller
{
    public function index()
    {
        $produtos = Produto::all();

        return view('produtos.index',['produtos'=>$produtos]);
    }

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

    public function store(ProdutoRequest $request)
    {
        $input = $request->all();
        Produto::create($input);

        return redirect('produtos');

    }
}
```

Dessa forma, toda requisição que chegar diretamente para a action store, será validada pelo método rules() da request ProdutoRequest.

Agora, se você tentar criar um novo produto pelo formulário e não informar nenhum valor, você perceberá que não será redirecionado para a listagem de produtos e, aparentemente, nada acontece.

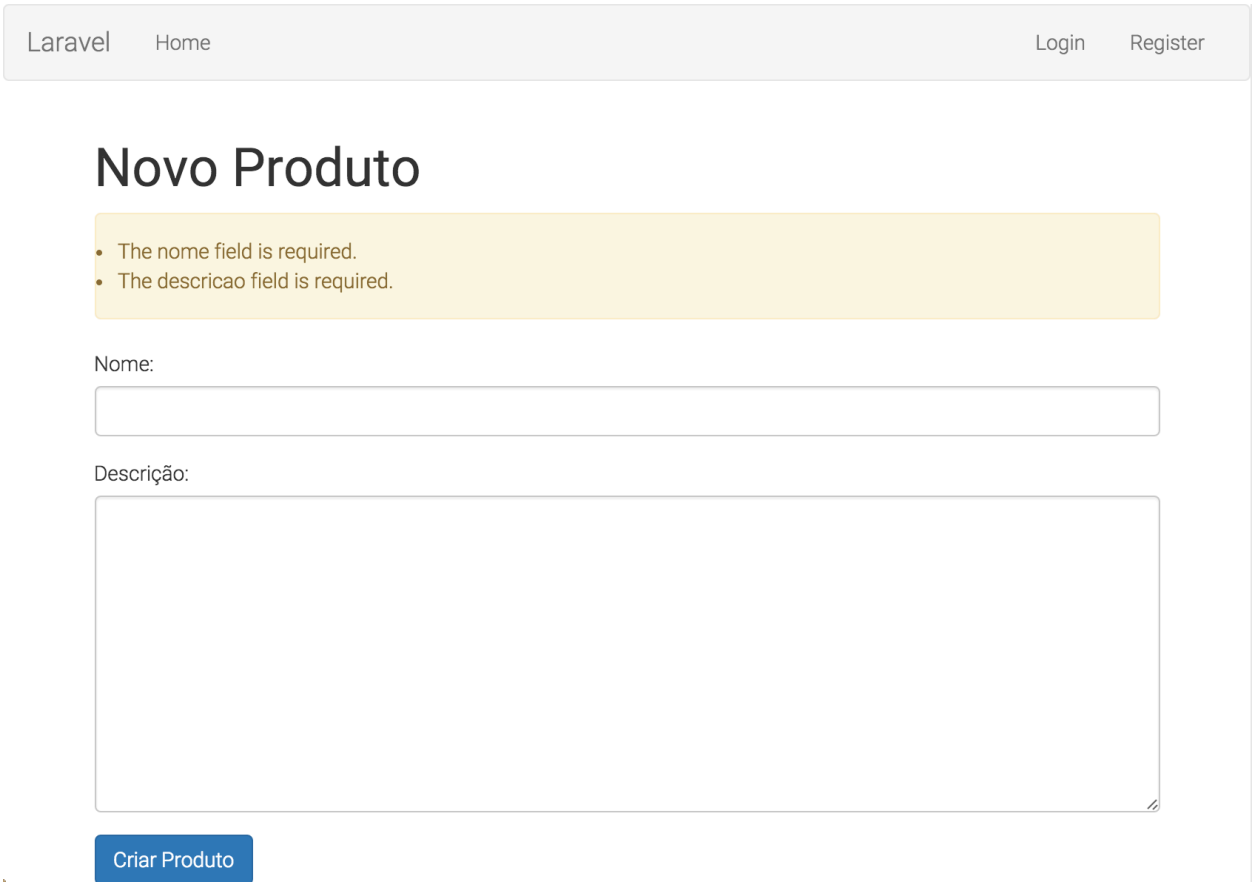
Só que na realidade, a nossa action *store* nem chegou a ser executada, uma vez que a requisição foi interrompida quando o ProdutoRequest fez a validação que não retornou de forma positiva.

Precisamos também informar ao usuário, quais informações estão incorretas e/ou faltantes, nesse caso, trabalharemos com a variável **$errors**, que armazena todos os erros de validação encontrados.

Adicionaremos o seguinte trecho de código no arquivo create.blade.php, verificando se há algum erro para ser exibido, caso positivo, exibiremos os erros em uma lista:

```
1 @if ($errors->any())
2         <ul class="alert alert-warning">
3                 @foreach($errors->all() as $error)
4                 <li>{{ $error }}</li>
5                 @endforeach
6         </ul>
7 @endif
```

O resultado ficará da seguinte forma:



Erro de validação do formulário

O código completo do arquivo create.blade.php ficou da seguinte forma:

```
1  @extends('app')
2
3  @section('content')
4      <div class="container">
5          <h1>Novo Produto</h1>
6
7          @if ($errors->any())
8              <ul class="alert alert-warning">
9                  @foreach($errors->all() as $error)
10                     <li>{{ $error }}</li>
11                 @endforeach
12             </ul>
13         @endif
14
15         {!! Form::open(['url'=>'produtos/store']) !!}
16
```

```
        <!-- Nome Form Input -->

        <div class="form-group">
            {!! Form::label('nome', 'Nome:') !!}
            {!! Form::text('nome', null, ['class'=>'form-control']) !!}
        </div>

        <!-- Descricao Form Input -->

        <div class="form-group">
            {!! Form::label('descricao', 'Descrição:') !!}
            {!! Form::textarea('descricao', null, ['class'=>'form-control']) !!}
        </div>

        <div class="form-group">
            {!! Form::submit('Criar Produto', ['class'=>'btn btn-primary']) !!}
        </div>

        {!! Form::close() !!}

    </div>
@endsection
```

Removendo registro

Trabalhar com remoção de um registro no Laravel é algo bem simples. Primeiramente, teremos que criar uma action específica para fazermos a remoção. Nesse caso, a chamaremos de destroy.

```
// ProdutosController.php
public function destroy($id)
{
    Produto::find($id)->delete();

    return redirect('produtos');
}

Simples assim!

Agora, temos apenas que adicionar isso a nossas rotas:

```php
```
```

```
// routes.php
Route::get('produtos/{id}/destroy','ProdutosController@destroy');
```

Quando você acessar por exemplo: http:://localhost:8000/produtos/1/destroy, ele fará, automaticamente, a exclusão do produto, cujo ID seja igual a 1.

Editando registro

O processo de edição de um registro é muito similar ao de alteração, principalmente se formos analisar a camada de visão.

O primeiro passo para fazer a edição de um registro, é entender o padrão das URIs que vamos utilizar.

```
// routes.php
Route::get('produtos/{id}/edit','ProdutosController@edit');
Route::put('produtos/{id}/update','ProdutosController@update');
```

O padrão da rota é muito similar ao do *destroy*, porém, estamos encaminhando o request para a action *edit*, para a mesma de trazer o formulário de edição, e a action *update* para efetivar a alteração no banco de dados.

Perceba que na rota de *update*, estamos utilizando o verbo http **PUT** ao invés de POST. Apesar de nossos browsers atualmente não suportarem tal recurso, o Laravel consegue contonar esse ponto com uma solução bem elegante.

Vamos agora criar a action *edit*:

```
public function edit($id)
{
    $produto = Produto::find($id);

    return view('produtos.edit', compact('produto'));
}
```

Veja que é um método extremamente simples, que pega o produto através do método find, e retorna o produto encontrado para nossa view.

Nesse ponto, temos apenas que criar nosso arquivo: edit.blade.php, dentro da pasta resources/views/produtos.

```
// resources/views/produtos/edit.blade.php

@extends('app')

@section('content')
    <div class="container">
        <h1>Editar Produto: {{$produto->name}}</h1>

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

        {!! Form::open(['url'=>"produtos/$produto->id/update", 'method'=>'put'])\
 !!}

        <!-- Nome Form Input -->

        <div class="form-group">
            {!! Form::label('nome', 'Nome:') !!}
            {!! Form::text('nome', $produto->nome, ['class'=>'form-control']) !!}
        </div>

        <!-- Descricao Form Input -->

        <div class="form-group">
            {!! Form::label('descricao', 'Descrição:') !!}
            {!! Form::textarea('descricao', $produto->nome, ['class'=>'form-cont\
rol']) !!}
        </div>

        <div class="form-group">
            {!! Form::submit('Salvar Produto', ['class'=>'btn btn-primary']) !!}
        </div>

        {!! Form::close() !!}

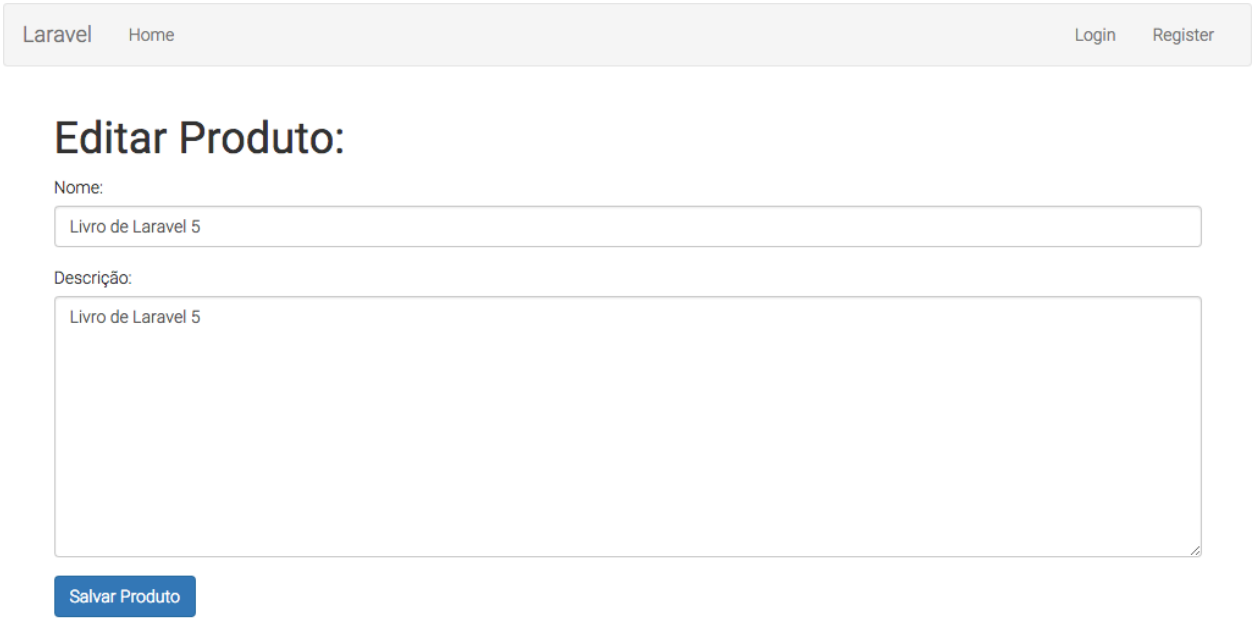
    </div>
@endsection
```

Nesse caso, fizemos poucas alterações em relação ao arquivo create.blade.php.

1. Alteramos o título da página para Editando Produto: Nome do produto
2. A action do form para: produtos/update
3. O Method do form de POST para PUT
4. O valor padrão dos campos dos formulários. Ex:

```
1  {!! Form::text('nome', $produto->nome, ['class'=>'form-control']) !!}
```

Veja que nesse caso, ao invés de trazermos o valor padrão do form como *null*, estamos trazendo com o valor do produto atual: *$produto->nome*.



Formulário de edição

Está lembrado que mudamos o method do form de POST para PUT? Veja abaixo, o código fonte gerado pelo formulário =)

Código fonte do form de edição

Perceba que o Laravel gerou automaticamente um campo hidden chamado **_method** com o valor de **PUT**. Apesar de nosso browser enviar a requisição via POST, o Laravel está aguardando uma requisição do tipo PUT, e nesse caso, ele fará a leitura do valor enviado pelo _method para se adequar.

Falta apenas implementarmos nosso método de update para realizarmos a alteração no banco de dados.

```
1  // ProdutosController.php
2
3  public function update(ProdutoRequest $request, $id)
4  {
5      $produto = Produto::find($id)->update($request->all());
6
7      return redirect('produtos');
8  }
```

Pronto, nosso update já está em funcionamento e nosso CRUD está completo.

Ajustando rotas

Até o momento, nosso arquivo de rotas está da seguinte forma:

```
Route::get('produtos','ProdutosController@index');

Route::get('produtos/create','ProdutosController@create');

Route::post('produtos/store','ProdutosController@store');

Route::get('produtos/{id}/destroy','ProdutosController@destroy');

Route::get('produtos/{id}/edit','ProdutosController@edit');

Route::put('produtos/{id}/update','ProdutosController@update');
```

O grande ponto nesse caso, é que podemos agrupar essas rotas por prefixo para facilitar a manutenção.

Vejamos:

```
Route::group(['prefix'=>'produtos'], function() {

    Route::get('','ProdutosController@index');

    Route::get('create','ProdutosController@create');

    Route::post('store','ProdutosController@store');

    Route::get('{id}/destroy','ProdutosController@destroy');

    Route::get('{id}/edit','ProdutosController@edit');

    Route::put('{id}/update','ProdutosController@update');

});
```

Veja que separamos tudo por prefixo, ou seja, se mudarmos o prefixo, automaticamente todas as nossas rotas mudarão.

De qualquer modo, há duas coisas que, no mínimo, ainda precisam ser melhoradas:

1. Nomear as rotas
2. Validar nosso parâmetro **id**

Nomeando nossas rotas

Antes de começarmos a nomear as rotas, você precisa entender quais são os infinitos benefícios que isso nos trás.

Quando trabalhamos com rotas nomeadas, todos os links de nosso sistema, ao invés de chamar uma URL fixa (como estamos fazendo atualmente), podem chamar uma rota nomeada, e nesse caso, o link é gerado dinamicamente de acordo com nossa rota; logo, se em algum momento resolvermos alterar nossas rotas, não precisaremos alterar absolutamente nada em nossas views.

Vamos fazer isso!

```
1  Route::group(['prefix'=>'produtos'], function() {
2
3      Route::get('',['as'=>'produtos', 'uses'=>'ProdutosController@index']);
4
5      Route::get('create',['as'=>'produtos.create', 'uses'=>'ProdutosController@cr\
6  eate']);
7
8      Route::post('store',['as'=>'produtos.store', 'uses'=>'ProdutosController@sto\
9  re']);
10
11     Route::get('{id}/destroy',['as'=>'produtos.destroy', 'uses'=>'ProdutosContro\
12 ller@destroy']);
13
14     Route::get('{id}/edit',['as'=>'produtos.edit', 'uses'=>'ProdutosController@e\
15 dit']);
16
17     Route::put('{id}/update',['as'=>'produtos.update', 'uses'=>'ProdutosControll\
18 er@update']);
19
20 });
```

Se você perceber, estamos utilizando uma *key* chamada **as** para nomearmos nossas rotas e a *key* **uses** para definirmos para qual controller/action a mesma será apontada.

Feito isso, basta agora alterarmos todos os locais onde forçamos uma URL por uma rota nomeada.

Veja como ficou nosso ProdutosController:

```php
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Produto;
use App\Http\Requests\ProdutoRequest;

class ProdutosController extends Controller
{
    public function index()
    {
        $produtos = Produto::all();

        return view('produtos.index',['produtos'=>$produtos]);
    }

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

    public function store(ProdutoRequest $request)
    {
        $input = $request->all();
        Produto::create($input);

        return redirect()->route('produtos');

    }

    public function edit($id)
    {
        $produto = Produto::find($id);

        return view('produtos.edit', compact('produto'));
    }

    public function update(ProdutoRequest $request, $id)
    {
        $produto = Produto::find($id)->update($request->all());

        return redirect()->route('produtos');
    }
```

```
43 }
```

Em todos os locais onde redirecionamos o usuário para /produtos, alteramos para: **redirect()->route('produtos');**

Também temos que alterar nossos forms: create.blade.php e edit.blade.php

```
1 // create.blade.php
2 {!! Form::open(['route'=>'produtos.store']) !!}
3
4 // edit.blade.php
5 {!! Form::open(['route'=>['produtos.update', $produto->id], 'method'=>'put']) !!}
```

Agora temos total flexibilidade de linkar qualquer recurso interno do sistema, sem ter a preocupação de nossas rotas mudarem no futuro.

Validando o parâmetro ID de nossas rotas

Um recurso que é de extrema importância no Laravel, é a possibilidade de podermos validar os parâmetros dinâmicos de nossa rota, dessa forma, podemos ter mais segurança dos tipos e formatos de dados que estão entrando em nossas requisições.

No nosso caso, utilizamos diversas vezes o parâmetro ID para identificar qual o produto em questão que desejamos realizar alguma ação.

Sabemos que nossos IDs são numéricos, logo, podemos simplesmente adicionar tal validação em nossa rota:

```
1 // routes.php
2
3 Route::group(['prefix'=>'produtos', 'where'=>['id'=>'[0-9]+']], function() {
4
5     Route::get('',['as'=>'produtos', 'uses'=>'ProdutosController@index']);
6
7     Route::get('create',['as'=>'produtos.create', 'uses'=>'ProdutosController@cr\
8 eate']);
9
10    Route::post('store',['as'=>'produtos.store', 'uses'=>'ProdutosController@sto\
11 re']);
12
13    Route::get('{id}/destroy',['as'=>'produtos.destroy', 'uses'=>'ProdutosContro\
14 ller@destroy']);
```

```
15
16      Route::get('{id}/edit',['as'=>'produtos.edit', 'uses'=>'ProdutosController@e\
17  dit']);
18
19      Route::put('{id}/update',['as'=>'produtos.update', 'uses'=>'ProdutosControll\
20  er@update']);
21
22  });
```

Se você tentar acessar qualquer URL que possua o parâmetro ID sem passar um valor numérico, perceberá que a rota não será encontrada.

Isso só acontece pelo fato de estarmos utilizando a *key* **where**, aplicando a regra de que apenas numéricos podem ser passados para o parâmetro *id* ([0-9]+).

Você pode fazer infinitos tipos de validações com suas rotas. Para saber mais, veja como isso pode ser aplicado: Documentação oficial do Laravel - Routing[5].

Linkando listagem de produtos

Para finalizar esse capítulo, faremos a linkagem desses recursos em nossa listagem de produtos no arquivo: index.blade.php:

1. Link para criarmos novos produtos
2. Link para editarmos um produto
3. Link para removermos um produto

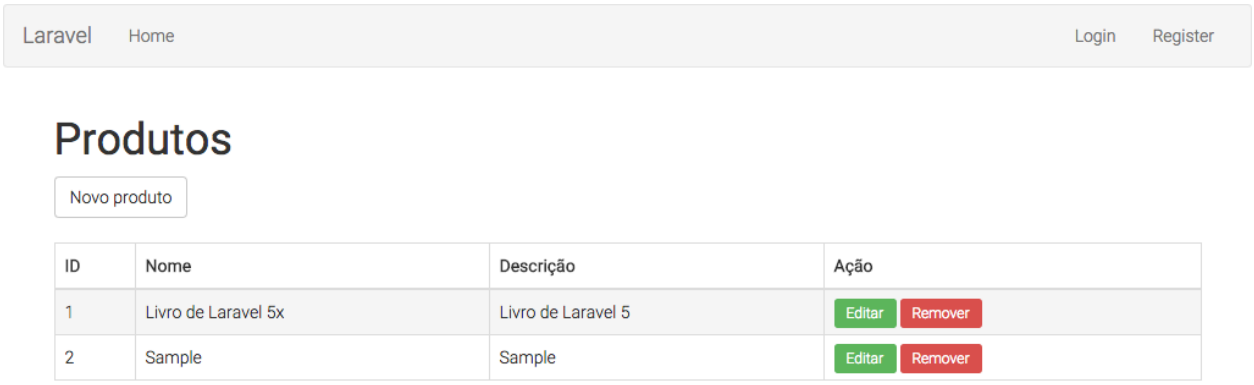
```
1   // index.blade.php
2
3   @extends('app')
4
5   @section('content')
6       <div class="container">
7           <h1>Produtos</h1>
8
9           <a href="{{ route('produtos.create') }}" class="btn btn-default">Novo pr\
10  oduto</a>
11          <br />
12          <br />
13          <table class="table table-striped table-bordered table-hover">
```

[5] http://laravel.com/docs/5.0/routing#route-parameters

```
14              <thead>
15              <tr>
16                  <th>ID</th>
17                  <th>Nome</th>
18                  <th>Descrição</th>
19                  <th>Ação</th>
20              </tr>
21              </thead>
22              <tbody>
23
24              @foreach($produtos as $produto)
25              <tr>
26                  <td>{{ $produto->id }}</td>
27                  <td>{{ $produto->nome }}</td>
28                  <td>{{ $produto->descricao }}</td>
29                  <td>
30                      <a href="{{ route('produtos.edit',['id'=>$produto->id]) }}" \
31  class="btn-sm btn-success">Editar</a>
32                      <a href="{{ route('produtos.destroy',['id'=>$produto->id]) }\
33  }" class="btn-sm btn-danger">Remover</a>
34                  </td>
35              </tr>
36              @endforeach
37
38              </tbody>
39          </table>
40      </div>
41  @endsection
```

Todos os recursos foram linkados utilizando o helper route(), onde o primeiro parâmetro é o nome de nossa rota, e o segundo (opcional), um array com os valores que a rota precisa (como o $id por exemplo). Ex: {{ route(‘produtos.create’) }}

Tivemos nesse caso, como resultado final:



Listagem de produtos

Relacionando Models

Um dos pontos que acredito que você já deva estar curioso, é como podemos fazer o relacionamento entre Models, trabalhando com o Eloquent no Laravel.

Acredite, trabalhar com relacionamentos é muito simples no Laravel, e é isso que vamos ver a partir de agora nesse capítulo.

Criando Model e Migration

No momento, possuímos nosso CRUD de produtos, todavia, gostaríamos de poder adicionar uma funcionalidade simples em nosso sistema: Avaliar produtos.

O objetivo é que para cada produto cadastrado, possamos criar diversas avaliações de clientes, logo, fica evidente que precisaremos de um outro participante que chamaremos de AvaliacaoProduto.

Nesse caso, vamos criar nosso Model:

```
1  php artisan make:model AvaliacaoProduto
2  Model created successfully.
3  Created Migration: 2015_05_06_180023_create_avaliacao_produtos_table
```

Com nosso model criado, precisaremos editar nosso arquivo de migração para que tenhamos o script de geração de nossa tabela.

```
1  <?php
2
3  use Illuminate\Database\Schema\Blueprint;
4  use Illuminate\Database\Migrations\Migration;
5
6  class CreateAvaliacaoProdutosTable extends Migration
7  {
8
9      /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14     public function up()
```

```
15      {
16          Schema::create('avaliacao_produtos', function (Blueprint $table) {
17              $table->increments('id');
18              $table->integer('produto_id')->unsigned();
19              $table->foreign('produto_id')->references('id')->on('produtos');
20              $table->smallInteger('nota');
21              $table->text('comentario');
22              $table->timestamps();
23          });
24      }
25
26      /**
27       * Reverse the migrations.
28       *
29       * @return void
30       */
31      public function down()
32      {
33          Schema::drop('avaliacao_produtos');
34      }
35
36  }
```

Perceba que nesse caso, além de criar os campos nota e comentário, também criamos um campo chamado produto_id e o definimos como chave estrangeira, fazendo relação com o id de nossa tabela de produtos:

```
1  $table->integer('produto_id');
2  $table->foreign('produto_id')->references('id')->on('produtos');
```

Agora, para que isso possa surtir efeito, temos que rodar nossas migrações novamente:

```
1  php artisan migrate
2  Migrated: 2015_05_06_180023_create_avaliacao_produtos_table
```

Protinho! Nosso banco de dados agora possui uma nova tabela chamada avaliacao_produtos, que possui uma chave estrangeira para nossa tabela de produtos.

Criando relacionamento

Agora que já possuímos nosso model AvaliacaoProduto, bem como sua tabela no banco de dados, precisaremos fazer nossos primeiros ajustes em nosso model para que ele possa se relacionar, "naturalmente", com nosso model Produto.

Para isso, inicialmente, adicionaremos o atributo $fillable em nosso model, para que possamos trabalhar com *mass assignment* e facilitar nosso processo de criação de novas avaliações.

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class AvaliacaoProduto extends Model
{
    protected $fillable = [
        'produto_id',
        'nota',
        'comentario'
    ];

}
```

Agora, para criarmos uma nova avaliação ficou muito simples.

Vamos fazer isso utilizando o tinker:

```
php artisan tinker
Psy Shell v0.4.4 (PHP 5.6.4 — cli) by Justin Hileman
>>> use App\AvaliacaoProduto;
=> false
>>> AvaliacaoProduto::create(['produto_id'=>1, 'nota'=>5, 'comentario'=>'primeir\
o comentario']);
=> <App\AvaliacaoProduto #000000004b2865de0000000159bd2d9f> {
       produto_id: 1,
       nota: 5,
       comentario: "primeiro comentario",
       updated_at: "2015-05-06 18:12:14",
       created_at: "2015-05-06 18:12:14",
       id: 1
   }
```

Perceba que criamos a avaliação de nosso produto com o ID **1**, de uma forma super tranquila, apenas executando o método *create()* e passando os atributos por um array associativo.

A pergunta que não quer calar:

Como faço descobrir qual é o produto referente a minha avaliação? Eu só tenho o ID!

Realmente, para fazer isso, primeiramente, precisamos entender um conceito muito simples. Vamos lá!

Se você perceber, um Produto pode ter diversas Avaliações, porém, uma avaliação pertence unicamente a um produto.

Nesse caso, definiremos essas relações, exatamente da forma:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class AvaliacaoProduto extends Model
{
    protected $fillable = [
        'produto_id',
        'nota',
        'comentario'
    ];

    public function produto()
    {
        return $this->belongsTo('App\Produto');
    }

}
```

Adicionamos um novo método em nosso Model chamado de *produto*, esse método possui um retorno bem sugestivo:

```
return $this->belongsTo('App\Produto');
```

Isto é, o meu model atual, pertence a um model de Produto, simples assim!

Com essa simples implementação, poderemos ter o seguinte resultado:

```
php artisan tinker
Psy Shell v0.4.4 (PHP 5.6.4 — cli) by Justin Hileman
>>> use App\AvaliacaoProduto;
=> false
>>> $avaliacao = AvaliacaoProduto::find(1);
=> <App\AvaliacaoProduto #000000006d7b045a00000013efd94ce> {
       id: "1",
       produto_id: "1",
       nota: "5",
       comentario: "primeiro comentario",
```

```
        created_at: "2015-05-06 18:12:14",
        updated_at: "2015-05-06 18:12:14"
    }
>>> $avaliacao->produto;
=> <App\Produto #000000006d7b0446000000013efd94ce> {
        id: "1",
        nome: "Livro de Laravel 5x",
        descricao: "Livro de Laravel 5",
        created_at: "2015-04-20 22:01:57",
        updated_at: "2015-04-28 20:42:34"
    }
>>> $avaliacao->produto->nome;
=> "Livro de Laravel 5x"
>>> $avaliacao->produto->descricao;
=> "Livro de Laravel 5"
```

Veja que agora, basta executarmos a chamada ->produto em nosso model, que simplesmente teremos acesso ao Model referente a nossa chave estrangeira.

Veja que estamos executando: $avaliacao->produto e não $avaliacao->produto().

Por outro lado, seria muito interessante podermos fazer o inverso, ou seja, através de um produto, podermos pegar todas as avaliações referenciadas a ele. Isso é totalmente possível, todavia, precisaremos criar um outro método para isso em nosso model Produto.

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Produto extends Model
{
    protected $fillable = ['nome', 'descricao'];

    public function avaliacoes()
    {
        return $this->hasMany('App\AvaliacaoProduto');
    }

}
```

Nesse caso, criamos um método chamado *avaliacoes()*, que tem o objetivo de retornar todas as avaliações referentes a um determinado produto.

Se você analisar mais a fundo, ao invés de utilizarmos *$this->belongsTo...*, como fizemos em nosso model AvaliacaoProduto, trabalhamos com *$this->hasMany...*, uma vez que o nosso produto pode ter diversas avaliações retornadas.

Vamos ver como ficou:

```
1  php artisan tinker
2  Psy Shell v0.4.4 (PHP 5.6.4 — cli) by Justin Hileman
3  >>> use App\Produto;
4  => false
5  >>> $produto = Produto::find(1);
6  => <App\Produto #000000002c9196ba00000017b59295e> {
7         id: "1",
8         nome: "Livro de Laravel 5x",
9         descricao: "Livro de Laravel 5",
10        created_at: "2015-04-20 22:01:57",
11        updated_at: "2015-04-28 20:42:34"
12    }
13 >>> $produto->avaliacoes;
14 => <Illuminate\Database\Eloquent\Collection #000000002c9196a000000017b59295e> [
15        <App\AvaliacaoProduto #000000002c9196a600000017b59295e> {
16            id: "1",
17            produto_id: "1",
18            nota: "5",
19            comentario: "primeiro comentario",
20            created_at: "2015-05-06 18:12:14",
21            updated_at: "2015-05-06 18:12:14"
22        }
23    ]
```

O código acima, nos deixa claro que ao chamarmos *avaliacoes* em nosso model Produto, automaticamente, ele fará a busca por todas as avaliações relacionadas e nos retornará uma *Collection* de models de ProdutoAvaliacao que facilmente poderá ser iterado com um *foreach*, por exemplo.

Tabela Pivot (ManyToMany)

Apesar dos relacionamentos anteriores serem extremamente utilizados, em diversos casos, precisamos criar relacionamentos com uma tabela intermediária, pois nesse caso, o mesmo será de MuitosParaMuitos (*ManyToMany*).

Para que possamos exemplificar melhor esse tipo de relacionamento, criaremos um novo Model chamado de Tag.

Nesse caso, poderemos atribuir diversas tags aos nossos produtos, logo, uma tag pode estar ligada a mais de um produto, da mesma forma que um produto poderá estar ligado a mais de uma tag.

```
php artisan make:model Tag
Model created successfully.
Created Migration: 2015_05_06_183634_create_tags_table
```

```
<?php

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

class CreateTagsTable extends Migration {

        /**
         * Run the migrations.
         *
         * @return void
         */
        public function up()
        {
                Schema::create('tags', function(Blueprint $table)
                {
                        $table->increments('id');
            $table->text('name');
                        $table->timestamps();
                });
        }

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

}
```

Agora que já possuímos nosso arquivo de migração para criarmos nossa tabela *tags*, basta, executarmos:

```
php artisan migrate
Migrated: 2015_05_06_183634_create_tags_table
```

Prontinho!

Já possuímos nosso Model Produto e Tag, porém, ainda está faltando nossa tabela para fazer a interligação entre esses dois participantes, nesse caso, vamos criar uma nova migration:

```
php artisan make:migration produtos_tags_table --create=produto_tag
Created Migration: 2015_05_06_184107_produtos_tags_table
```

O parâmetro –create, informa ao Laravel para criar o arquivo de migração preparado para o processo de criação de uma tabela de banco de dados. Se você pretende fazer a alteração de uma tabela, basta passar o parâmetro: –table=nome-da-tabela.

```php
<?php

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

class ProdutosTagsTable extends Migration
{

    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('produto_tag', function (Blueprint $table) {
            $table->increments('id');
            $table->integer('produto_id');
            $table->foreign('produto_id')->references('id')->on('produtos');
            $table->integer('tag_id');
            $table->foreign('tag_id')->references('id')->on('tags');
        });
```

```
23      }
24
25      /**
26       * Reverse the migrations.
27       *
28       * @return void
29       */
30      public function down()
31      {
32          Schema::drop('produtos_tags');
33      }
34
35  }
```

Nessa tabela, criamos duas chaves estrangeiras: produto_id e tag_id, dessa forma, será possível fazer a ligação ManyToMany.

```
1  php artisan migrate
2  Migrated: 2015_05_06_184107_produtos_tags_table
```

Da mesma forma que tivemos que implementar métodos em nossos models para criarmos as relações entre Produto e AvaliacaoProduto, também, teremos que fazer essa implementação para o relacionamento entre Produto e Tag.

```
1  <?php namespace App;
2
3  use Illuminate\Database\Eloquent\Model;
4
5  class Produto extends Model
6  {
7      protected $fillable = ['nome', 'descricao'];
8
9      public function avaliacoes()
10     {
11         return $this->hasMany('App\AvaliacaoProduto');
12     }
13
14     public function tags()
15     {
16         return $this->belongsToMany('App\Tag');
17     }
18
19 }
```

Veja que ao implementarmos o método *tag()*, retornamos $this->belongsToMany ao invés de simplesmente $this->belongsTo. Isso já é o suficiente para trabalharmos com o relacionamento de muitos-para-muitos. Vamos ao tinker:

```
1  php artisan tinker
2  Psy Shell v0.4.4 (PHP 5.6.4 — cli) by Justin Hileman
3  >>> use App\Tag;
4  => false
5  >>> use App\Produto;
6  => false
7  >>> $tag = new Tag;
8  => <App\Tag #000000002f242af300000001685d9118> {}
9  >>> $tag->name = "Tag 1";
10 => "Tag 1"
11 >>> $tag->save()
12 => true
13 >>> $produto = Produto::find(1);
14 => <App\Produto #000000002f242ae700000001685d9118> {
15        id: "1",
16        nome: "Livro de Laravel 5x",
17        descricao: "Livro de Laravel 5",
18        created_at: "2015-04-20 22:01:57",
19        updated_at: "2015-04-28 20:42:34"
20    }
21 >>> $produto->tags()->attach($tag);
22 => null
23 >>> $produto->tags;
24 => <Illuminate\Database\Eloquent\Collection #000000002f242ae500000001685d9118> [
25        <App\Tag #000000002f242ae000000001685d9118> {
26            id: "1",
27            name: "Tag 1",
28            created_at: "2015-05-06 18:53:55",
29            updated_at: "2015-05-06 18:53:55",
30            pivot: <Illuminate\Database\Eloquent\Relations\Pivot #000000002f242af\
31 b00000001685d9118> {
32                produto_id: "1",
33                tag_id: "1"
34            }
35        }
36    ]
```

No exemplo acima, criamos uma tag chamada de *Tag 1* e a *atachamos* ao nosso produto utilizando: $produto->tags()->attach($tag).

Ao chamarmos $produto->tag, recebemos normalmente uma coleção de tags que está vinculada ao nosso produto.

Tenha mais detalhes sobre os relacionamentos / associações de Models do Laravel, em sua Documentação Oficial[6].

[6] http://laravel.com/docs/5.1/eloquent-relationships

Container de serviços

Serviços

Um dos conceitos mais importantes que podemos aprender para trabalhar com qualquer framework moderno é o conceito de serviços.

Serviços podem ser definidos como biblioteca(s), que normalmente são utilizadas em diversas partes do nosso software.

O grande ponto, é que muitas dessas bibliotecas, podem possuir complexidades para sua configuração inicial.

Deixe-me lhe dar um exemplo:

Vamos supor que temos um serviço chamado de SampleService que para ser configurado, depende de um segundo serviço chamado DependencyService.

```
1  class MinhaClasse
2  {
3
4      public function __construct()
5      {
6          $config = [
7              'config1' => 1,
8              'config2' => 2
9          ];
10
11         $dependencyService = new DependencyService($config);
12
13         $sampleService = new SampleService($dependencyService);
14     }
15 }
```

Agora, vamos imaginar que precisamos usar essa biblioteca em diversas partes de nosso software.

Nesse ponto, acredito que você já consiga perceber que todas as vezes que precisarmos dessa bibliteca, teremos que realizar essas configurações; e eventualmente, se essas configurações forem alteradas, teremos que mudar em diversos lugares de nossa aplicação.

Um outro grande problema de trabalharmos com tal abordagem, é o acoplamento gerado na classe que utilizará essa bibliteca, nesse caso a classe MinhaClasse.

Essa classe depende diretamente das classes DependencyService e SampleService, logo, futuramente, para que possamos reaproveitá-la em outro software, etc teremos que, obrigatoriamente, levar essas dependências também.

Lembre-se, quanto mais desacopladas forem nossas classes, melhor!

Dependency Injection

A melhor forma que podemos encontrar para termos classes mais desacopladas, é trabalhar com um conceito muito simples de injeção de dependência, mais conhecido como DI ou Dependency Injection.

No caso acima, você pode perceber que as nossas classes de serviço estão sendo instanciadas dentro da classe MinhaClasse.

Vamos mudar isso!

```
1  // MinhaClasse.php
2  <?php
3  class MinhaClasse
4  {
5
6      private $sampleService;
7
8      public function __construct(SampleService $sampleService)
9      {
10
11         $this->sampleService = $sampleService;
12
13     }
14 }
15
16 //bootstrap.php
17 <?php
18
19 $config = [
20     'config1' => 1,
21     'config2' => 2
22 ];
23
24 $dependencyService = new DependencyService($config);
25
26 $sampleService = new SampleService($dependencyService);
```

```
27
28 $minhaClasse = new MinhaClasse($sampleService);
```

Melhoramos nossa a classe MinhaClasse, uma vez que não estamos instanciando objetos dentro dela mesma, todavia, ainda temos um probleminha.

PROGRAME SEMPRE PARA UMA INTERFACE E NUNCA PARA UMA IMPLEMENTAÇÃO

O que a dica acima está dizendo, é que mesmo injetanto nossa depedência no construtor da classe, a mesma ainda depende, diretamente, de uma implementação concreta de SampleService.

Isso não é adequado, pois se algum dia quisermos fazer a classe MinhaClasse utilizar um outro serviço similar ao SampleService, não será possível.

A forma mais correta de sairmos desse impasse, é trabalharmos com interfaces, fazendo com que nossa classe SampleService siga um determinado contrato para sua implementação.

```
1  <?php
2  // SampleServiceInterface.php
3  interface SampleServiceInterface
4  {
5  }
6
7
8  // SampleService.php
9  <?php
10
11 class SampleService implements SampleServiceInterface
12 {
13 }
14
15 // MinhaClasse.php
16 <?php
17 class MinhaClasse
18 {
19     private $sampleService;
20
21     public function __construct(SampleServiceInterface $sampleService)
22     {
23
24         $this->sampleService = $sampleService;
25
```

```
26      }
27  }
28
29  //bootstrap.php
30  <?php
31
32  $config = [
33      'config1' => 1,
34      'config2' => 2
35  ];
36
37  $dependencyService = new DependencyService($config);
38
39  $sampleService = new SampleService($dependencyService);
40
41  $minhaClasse = new MinhaClasse($sampleService);
```

Agora sim! Se em algum momento quisermos passar um outro tipo de SampleService para a classe MinhaClasse, basta essa classe implementar a interface SampleServiceInterface.

Containers de serviços

Tudo que acabei de lhe apresentar, foram conceitos básicos de orientação a objetos, porém, nossa principal questão ainda permanece: Como podemos gerenciar todos esses nossos serviços, desde seu processo de instanciação, até a utilização dos mesmos dentro de nosso software?

A resposta para isso é: Container de Serviços, ou DiC (Dependency Injection Container).

Um container de serviços, tem o objetivo de criar, armazenar e disponibilizar os serviços de nossas aplicações.

Container de serviços do Laravel

No caso do Laravel, contamos com um DiC extremamente simples de ser utilizado.

Para facilitar minha explicação, vamos criar uma classe chamada: ProdutoRepository, a mesma terá o objetivo de ter os principais métodos para executarmos nossas operações com produtos.

Essa classe, também nos ajudará a não ficarmos chamando nosso model diretamente, nesse caso, se um dia quisermos trocar nosso ORM, não teremos mudanças expressivas em outras camadas de nossa aplicação.

Nosso objetivo nesse momento é deixarmos nossa aplicação o mais desacoplada possível, e que possamos facilmente trocar a implementação concreta de nossa classe ProdutoRepository.

Apenas lembrando que faremos uma implementação de um Repository de uma forma MUITO simples, apenas para que você consiga entender o quão flexível podemos deixar nossa aplicação.

Nossa classe ProdutoRepository de forma completa se parece com isso:

```php
// app/Repositories/Eloquent/ProdutoRepository.php
<?php namespace App\Repositories\Eloquent;

use App\Produto;

class ProdutoRepository
{
    /**
     * @param Produto $model
     */
    public function __construct(Produto $model)
    {
        /** @var Produto $model */
        $this->model = $model;
    }

    public function getAll()
    {
        return $this->model->all();
    }

    public function find($id)
    {
        return $this->model->find($id);
    }

    public function create(array $data)
    {
        return $this->model->create($data);
    }

    public function update($id, array $data)
    {
        return $this->model->find($id)->update($data);
    }

```

```
37      public function delete($id)
38      {
39          return $this->model->find($id)->delete();
40      }
41  }
```

Se você perceber, o código gera uma camada acima de nosso model, para que não precisemos expor nossos models diretamente em nossas operações com banco de dados em nossos controllers e outras classes de serviços.

O grande problema no código acima, é que uma vez que usamos uma implementação concreta de uma classe, todo nosso software ficará extremamente acoplado a essa implementação.

Vamos imaginar se amanhã, desejássemos trocar esse repository por um repository do Doctrine ORM, por exemplo. Certamente teríamos sérios problemas, pois teríamos que sair alterando em todos os lugares que essa classe concreta está sendo chamada.

Uma forma para deixarmos isso mais flexível, é fazer com que nossa classe implemente uma interface. Talvez no momento, isso não faça tanto sentido assim, mas tentarei ser o mais prático possível nesse caso.

Vamos lá:

Criando uma Interface e um Abastract Repository

O primeiro passo nessa nossa refatoração, será definirmos uma Interface (contrato) para todos os repositories que o implementarem.

```
1   // app/Repositories/Contracts/RepositoryInterface.php
2
3   <?php namespace App\Repositories\Contracts;
4
5   interface RepositoryInterface
6   {
7       public function getAll();
8       public function find($id);
9       public function create(array $data);
10      public function update($id, array $data);
11      public function delete($id);
12  }
```

Agora está muito claro quais são os métodos que todos os nossos repositories, no mínimo devam implementar.

Nesse momento, podemos criar uma classe Abstrata para que nossos repositories possam estender.

```php
// app/Repositories/AbstractoRepository.php
<?php namespace App\Repositories;

abstract class AbstractRepository implements Contracts\RepositoryInterface
{
    protected $model;

    public function getAll()
    {
        return $this->model->all();
    }

    public function find($id)
    {
        return $this->model->find($id);
    }

    public function create(array $data)
    {
        return $this->model->create($data);
    }

    public function update($id, array $data)
    {
        return $this->model->find($id)->update($data);
    }

    public function delete($id)
    {
        return $this->model->find($id)->delete();
    }
}
```

Prontinho, agora para que temos um repository funcionando, basta o mesmo estender de AbstractRepository.

Vamos ver isso na prática:

```php
<?php namespace App\Repositories\Eloquent;

use App\Produto;
use App\Repositories\AbstractRepository;

class ProdutoRepository extends AbstractRepository
{
    /**
     * @param Produto $model
     */
    public function __construct(Produto $model)
    {
        /** @var Produto $model */
        $this->model = $model;
    }
}
```

Ótimo! Agora nossa classe ProdutoRepository está muito mais simplificada, todavia, ainda temos um sério problema com ela. Se sairmos utilizando essa implementação concreta de ProdutoRepository, ainda teremos aquele grande problema de acoplamento, uma vez que todas as classes que a utilizarem, dependerão, exclusivamente, dessa implementação concreta.

Nesse ponto, para resolvermos esse problema, basta criarmos uma interface específica para o ProdutoRepository, dessa forma, todas as classes agora precisão apenas da da Interface do ProdutoRepository e não mais de sua implementação concreta.

```php
// app/Repositories/Contracts/ProdutoRepositoryInterface
<?php  namespace App\Repositories\Contracts;

interface ProdutoRepositoryInterface
{
    // aqui vão os métodos específicos de nosso repository de produto
}
```

```php
<?php namespace App\Repositories\Eloquent;

use App\Produto;
use App\Repositories\AbstractRepository;
use App\Repositories\Contracts\ProdutoRepositoryInterface;

class ProdutoRepository extends AbstractRepository implements ProdutoRepositoryI\
nterface
{
    /**
     * @param Produto $model
     */
    public function __construct(Produto $model)
    {
        /** @var Produto $model */
        $this->model = $model;
    }
}
```

Missão cumprida!

Agora, nós temos uma classe ProdutoRepository que possui uma Interface chamada ProdutoRepositoryInterface, logo, qualquer referência que façamos em nosso software para o ProdutoRepository, na realidade será feita através de sua interface.

Vejamos um exemplo em nosso ProdutosController:

```php
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Repositories\Contracts\ProdutoRepositoryInterface;

class ProdutosController extends Controller
{

    private $produtoRepository;

    public function __construct(ProdutoRepositoryInterface $produtoRepository)
    {
        $this->produtoRepository = $produtoRepository;
    }

    public function index()
```

```
17      {
18          $produtos = $this->produtoRepository->getAll();
19
20          return view('produtos.index', ['produtos' => $produtos]);
21      }
22  }
```

Perceba que não estamos fazendo em momento algum referência para nossa classe ProdutoRepository, mas sim para nossa Interface.

A pergunta que não quer calar é: Como o Laravel saberá que dada a Interface **ProdutoRepositoryInterface** ele deverá instanciar a classe concreta **ProdutoRepository**?

Na realidade, ele não sabe, mas nesse caso, informaremos isso para ele em seu Container de Serviços. Para isso vamos registrar isso no AppServiceProvider.

```
1   // app/Providers/AppServiceProvider.php
2   <?php namespace App\Providers;
3
4   use Illuminate\Support\ServiceProvider;
5
6   class AppServiceProvider extends ServiceProvider
7   {
8
9       /**
10       * Bootstrap any application services.
11       *
12       * @return void
13       */
14      public function boot()
15      {
16          //
17      }
18
19      /**
20       * Register any application services.
21       *
22       * This service provider is a great spot to register your various container
23       * bindings with the application. As you can see, we are registering our
24       * "Registrar" implementation here. You can add your own bindings too!
25       *
26       * @return void
27       */
```

```
28      public function register()
29      {
30          $this->app->bind(
31              'Illuminate\Contracts\Auth\Registrar',
32              'App\Services\Registrar'
33          );
34
35          $this->app->bind(
36              'App\Repositories\Contracts\ProdutoRepositoryInterface',
37              'App\Repositories\Eloquent\ProdutoRepository'
38          );
39      }
40
41  }
```

Nesse caso, estamos dizendo para o container que todas as vezes que a Interface: *AppRepositoriesContractsProdutoRepositoryInterface* for injetada como dependência, automaticamente o Laravel deverá retornar a classe concreta: *AppRepositoriesEloquentProdutoRepository*.

Isso nos gerou uma flexibilidade fantástica, pois em se algum momento quisermos trocar nosso repository, basta mudarmos a chamada desse ServiceProvider e outra classe concreta que implemente essa mesma interface será retornada.

Por exemplo:

```
1  $this->app->bind(
2      'App\Repositories\Contracts\ProdutoRepositoryInterface',
3      'App\Repositories\DoctrineORM\ProdutoRepository'
4  );
```

Perceba que nesse caso hipotético, se chamarmos a interface *ProdutoRepositoryInterface*, agora teremos o ProdutoRepository do DoctrineORM, por exemplo. Também não teremos que mudar uma virgula em nosso controller ou em outras classes.

Outros recursos do Container do Laravel

Binding

Muitas vezes, precisamos registrar um serviço no container do Laravel, nesse caso, o método *bind()*, como você já viu acima é o essencial.

Mas ver alguns outros exemplos:

```php
$this->app->bind('NomeDoServico', function($app) {
    return new \Classe\Xpto(new \Classe DependenciaXpto));
});
```

Nesse caso, sempre que você precisar do serviço, basta você fazer a chamada para invocá-lo:

```php
$this->app->make('NomeDoServico');
// ou
app()->make('NomeDoServico'); // Utilizando helper
```

Perceba que nesse caso, se em algum momento você precisar mudar a fabricação de seu serviço, você fará isso apenas uma vez.

Singleton

Calma calma! Eu sei que o nome assusta, principalmente pelo fato de que muitas vezes, o Singleton pode ser considerado um anti-pattern. Essa forma de registrar um serviço, tem apenas o objetivo de garantir que teremos apenas uma instância do serviço a ser chamado, ou seja, podemos utilizar o *$this->app->make('Servico')* quantas vezes quisermos, que receberemos sempre o mesmo objeto (referência).

```php
$this->app->singleton('NomeDoServico', function($app) {
    return new \Classe\Xpto(new \Classe DependenciaXpto));
});
```

Instances

Em determinadas situações, já podemos ter a instância do serviço a ser registrado, nesse caso, podemos simplesmente fazer o registro, passando o objeto como parâmetro:

```php
$servico = new Servico(new Dependencia);
$this->app->instance('NomeDoServico', $servico);
```

Com essas opções, você será capaz de registrar suas classes, dependências e serviços para que você possa ter acesso as mesmas em qualquer lugar de sua aplicação Laravel. Sem dúvidas, esse é um dos recursos mais essenciais que podemos ter no framework para mantermos nossa aplicação da forma mais desacoplada possível.

Autenticação

Um dos pontos que provavelmente você já deva estar curioso, é sobre como funciona o processo de autenticação do Laravel.

Uma coisa que posso lhe dizer, é que você ficará surpreso do quão fácil é todo esse processo, nesse ponto, vamos começar com os conceitos básicos.

Forçando a autenticação para um usuário

Vamos supor que temos um usuário chamado: Wesley com o ID = 1 em nossa tabela de users.

Lembre-se, quando rodando nossas migrations, o Laravel por padrão já cria essa tabela para nós.

Por hora, não trabalharemos em um Controller, mas sim no arquivo de rotas para deixarmos nosso exemplo o mais simplificado e didático possível.

```
1  <?php
2  // app/Http/routes.php
3
4  // .... Outras rotas...
5
6  Route::get('auth/login', function() {
7
8      $user = \App\User::find(1);
9      Auth::login($user);
10
11 });
```

Pronto! Apenas isso!

Quando acessarmos a /auth/login, automaticamente o sistema irá autenticar o usuário, cujo ID = 1.

Perceba que estamos utilizando uma Facade do Laravel chamada de *Auth*. A mesma é capaz de executar todos os comandos referente ao processo de autenticação, inclusive, autenticar um determinado usuário sem a necessidade de saber a senha do mesmo.

Agora, como podemos saber se realmente o que fizemos surtiu efeito?

```php
<?php
// app/Http/routes.php

// .... Outras rotas...

Route:get('auth/login', function() {

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

    if(Auth::check() {
       return "logado!";
    }

});
```

Ao utilizarmos o método *Auth::check()*, pudemos verificar se o usuário está logado no sistema, retornando assim a string *logado*.

> Resumo da ópera: Todas as vezes que você quiser verificar se um usuário está logado no sistema, utilize o método *check()*.

Realizando logout

Realizar o logout é tão simples quanto fazer a própria autenticação "forçada" de um determinado usuário.

Vamos então criar uma nova rota para que o logout seja implementado.

```php
<?php
// app/Http/routes.php

// .... Outras rotas...

Route:get('auth/login', function() {

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

    if(Auth::check() {
       return "logado!";
```

```
    }

});

Route::get('auth/logout', function() {

    Auth::logout();

});
```

Sério mesmo que é apenas isso. Dessa forma, você terá realizado o logout do usuário no sistema.

Autenticando usuário com login e senha

Agora que você já aprendeu a autenticar um usuário de forma simples e também realizar o processo de logout, vamos aprender o processo de autenticação mais comum, onde entramos com um username (que nesse caso será nosso email) e uma determinada senha.

Vamos supor que a senha correta de nosso usuário senha = 123456.

```
<?php
// app/Http/routes.php

// .... Outras rotas...

Route:get('auth/login', function() {

    if(Auth::attempt(['email'=> 'eu@eu.com', 'password'=>123456])) {
        return "logado!";
    }
});
```

Se você perceber, apenas trocamos o método *login()* pelo método *attempt* que recebeu um array com as credenciais do usuário e como nesse caso as mesmas estão corretas, o usuário já foi autenticado diretamente no sistema.

Utilizando AuthController

Para que não tenhamos que ficar criando todos esse processo de autenticação manualmente, etc. O Laravel nos provê um controller chamado de AuthAuthController, o mesmo possui todos os métodos

para autenticar, realizar logout, criar um novo usuário, etc, logo, vejamos agora como trabalhar com tudo isso que já vem pronto ao nosso favor.

Primeiramente, criaremos um arquivo chamado login.blade.php dentro da pasta resources/views/auth (você também terá de criar a pasta auth).

```
<!-- resources/views/auth/login.blade.php -->

<form method="POST" action="/auth/login">
    {!! csrf_field() !!}

    <div>
        Email
        <input type="email" name="email" value="{{ old('email') }}">
    </div>

    <div>
        Password
        <input type="password" name="password" id="password">
    </div>

    <div>
        <input type="checkbox" name="remember"> Remember Me
    </div>

    <div>
        <button type="submit">Login</button>
    </div>
</form>
```

No arquivo de rotas, utilizaremos a seguinte:

```
<?php
// app/Http/routes.php

// .... Outras rotas...

Route:get('auth/login', 'Auth\AuthController@getLogin');
```

Perceba que ao acessar a auth/login, você verá o formulário de autenticação que utilizamos, porém, se você tentar submter o mesmo, um erro será exibido, pois ainda não configuramos a rota POST para esse recurso.

Vamos lá:

```php
<?php
// app/Http/routes.php

// .... Outras rotas...

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

Agora, quando você submeter o formulário, a action *postLogin* será executada e fará seu processo de autenticação.

Na documentação oficial do framework, você poderá pegar exemplos de templates para trabalhar com o processo de criar um novo usuário, entre outros. http://laravel.com/docs/5.1/authentication[7]

A dica final que posso lhe dar sobre todo o processo de autenticação, criação de usuários, recuperação de senha, entre outros é utilizar as rotas do tipo *controllers*.

```php
<?php
// app/Http/routes.php

// .... Outras rotas...

Route::controllers([
    'auth' => 'Auth\AuthController',
    'password' => 'Auth\PasswordController',
]);
```

Com esse tipo de rota, as actions que começam com get, como getLogin, ou seja: /auth/login, getLogout como auth/logout, serão reconhecidas automaticamente, logo, você não precisará declarar todas as ações no arquivo de rotas.

Apeans lembrando que o mesmo vale para as action que iniciam com post, como postLogin, que se refere ao método http POST para auth/login, entre outros.

Protegendo uma rota

Uma vez que já temos nosso processo de login simples, provavelmente você desejará proteger determinadas rotas para que apenas usuários autenticados possam acessá-la.

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

Para fazer isso, utilizaremos o Middleware de autenticação do Laravel

Middlewares são classes que interceptam nossas requests, fazem um processamento e deixam o framework seguir com seu fluxo. No caso do Middlware de autenticação, ele verificar se o usuário está logado antes de seguir com o fluxo normal do framework.

Vamos imaginar que estamos querendo acessar a rota: /secreto.

```
1  <?php
2  // app/Http/routes.php
3
4  // .... Outras rotas...
5
6  Route::get('secreto', ['middleware'=>'auth', function() {
7
8          return "Conteudo visivel apenas para usuários autenticados";
9
10 }];
```

Nesse caso, quando tentamos acessar a rota secreto, o middleware de autenticação é chamado e verifica se o usuário está autenticado, caso esteja, ele deixa a aplicação seguir normalmente, caso negativo, encaminhará automaticamente o usuário para a rota de login.

Não deixe de conhecer meu mais novo curso online de **Laravel com AngularJS** fazendo o deploy autoescalável na Amazon AWS. http://sites.code.education/laravel-com-angularjs/[8]

Continua...

[8] http://sites.code.education/laravel-com-angularjs/