Laravel 5.6 – Framework Web for Artisans

Gabriel Marques de Melo

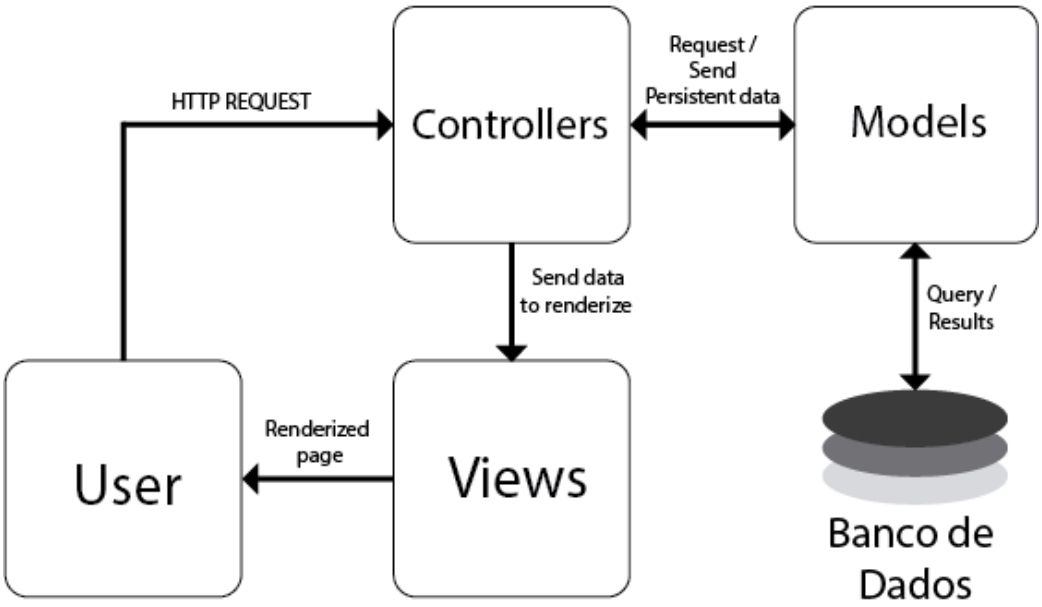
1. INTRODUÇÃO

O Laravel é um framework web PHP, baseado na arquitetura MVC, que busca agilizar e facilitar a criação de aplicações, evitando a codificação repetitiva e prezando por boas práticas e seguindo padrões de design. Seus diferenciais estão na intuitividade da sua estrutura de arquivos e diretórios, na sua rica documentação e, principalmente, na sua grande e participativa comunidade – não me deparei com nenhuma dúvida em que já não existisse um tópico respondido nos *Laracasts*[1] ou mesmo no professor *Stack overflow*.

A ideia deste documento é introduzir, superficialmente, os aspectos e funcionamentos do framework, bem como conduzir os leitores à demais documentos e referências.

1.1. MVC (Model-view-controller):

O MVC é um padrão de arquitetura de software que tem como objetivo isolar a representação da informação da interação do usuário.



[1] *laracasts.com/discuss*

1.1.1. Model

As models são classes que modelam uma entidade e a torna acessível para a aplicação de uma forma dinâmica. Podemos dizer que são usadas como uma interface entre o banco de dados e a aplicação.

1.1.2. View

As views são representações - visuais - dos dados de uma aplicação, voltados para um ou mais usuários.

1.1.3. Controller

Geralmente, os componentes mais complexos, em termos de codificação, em uma aplicação de arquitetura MVC são os controladores[2]. Estes componentes são classes que organizam a logica de uma ou mais rotas. As rotas encaminham a aplicação para o controlador, que, por sua vez, computa as operações necessárias à lógica da aplicação, podendo acessar as models para obter dados persistentes, e direciona uma view, passando os parâmetros necessários para sua renderização.

1.2. Design pattern FACADE

Muito usadas na estrutura do Laravel, as Facades, aqui citadas, são classes que seguem o padrão de design Facade.

O Facade é uma camada extra que acessa bibliotecas externas, isolando-as da nossa aplicação. Ela concentra todas as chamadas das classes de um serviço, tornando a manutenção do software mais fácil, uma vez que não temos controle sobre a atualização das bibliotecas de terceiros.

Veremos várias Facades neste documento. Dentre elas, podemos citar a *Auth*, *DB* e a *Session*.

1.3. Closures

Closures são classes internas do PHP que instanciam as funções anônimas nesta linguagem. As funções anônimas possuem características diferentes das funções convencionais:

[2] Na verdade, uma boa prática de desenvolvimento é utilizar os controladores como responsáveis apenas das requisições HTTP, uma vez que existem outras maneiras de uma aplicação receber solicitações, como *cren jobs*, *queue jobs, etc.*

- Não possuem um nome definido;
- Podem ser atreladas à uma variável. Caso a variável não esteja disponível no escopo, a função será, então, inalcançável;
- Podem ser atribuídas e passadas como parâmetros de funções/métodos;

Por exemplo, abaixo temos uma variável que recebe o valor retornado pela Closure:

```php
$soma = function ($a, $b) {
        return $a + $b;
};
```

2. Preparação do Ambiente

2.1. Windows:

- Homestead/vagrant

FAZENDO...

- Configuração do Homestead.yaml

```
ip: "192.168.10.10"
memory: 2048
cpus: 1
provider: virtualbox

authorize: c:/Users/melo/.ssh/id_rsa.pub

keys:
    - c:/Users/melo/.ssh/id_rsa

# Mapeamento da pasta local com pasta da vm
folders:
    - map: d:/Homestead_projects
      to: /home/vagrant/code

# Mapeamento do acesso aos aplicativos
sites:
    - map: homestead.test
      to: /home/vagrant/code/Laravel/public

databases:
    - homestead

# blackfire:
#     - id: foo
#       token: bar
#       client-id: foo
#       client-token: bar

# ports:
#     - send: 50000
#       to: 5000
#     - send: 7777
#       to: 777
#       protocol: udp
```

- Script para rodar *vangrant up* como *homestead up* e *homestead ssh* de qualquer lugar;

2.2. Linux

2.2.1. Pré-requisitos

2.2.1.1. Instalando Apache e PHP 7.2

```
sudo add-apt-repository ppa:ondrej/php
sudo apt-get update
sudo apt-get install apache2 libapache2-mod-php7.2 php7.2
php7.2-xml php7.2-gd php7.2-opcache php7.2-mbstring
```

2.2.1.2. Instalando o Composer (gerenciador de pacotes PHP)

```
cd /tmp
curl -sS https://getcomposer.org/installer | php
sudo mv composer.phar /usr/local/bin/composer
```

2.2.2. Criando um novo projeto Laravel

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

2.2.3. Configurando o apache

2.2.3.1. Permissões de acesso

```
sudo chgrp -R www-data /var/www/html/Nome-projeto
sudo chmod -R 775 /var/www/html/Nome-projeto/storage
```

2.2.3.2. Arquivo de configuração do projeto

```
cd /etc/apache2/sites-available
sudo nano laravel.conf
```

Exemplo de arquivo de configuração

```
<VirtualHost *:80>
    ServerName seuDominio.app

    ServerAdmin webmaster@localhost
    DocumentRoot /var/www/html/Nome-projeto/public

    <Directory /var/www/html/Nome-projeto>
        AllowOverride All
    </Directory>

    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

2.2.3.3. Desabilitando virtualhost padrão e habilitando o novo

```
sudo a2dissite 000-default.conf
sudo a2ensite laravel.conf
sudo service apache2 reload
```

2.2.4. Configurações iniciais no projeto

2.2.4.1. Gerando uma chave de aplicação

```
php artisan key:generate
```

Se a chave não é declarada, suas sessões de usuário e outros dados criptografados não estarão seguros!

2.2.4.2. Configuração do(s) banco de dados

As configurações do banco de dados são feitas no arquivo *config/database.php*. O arquivo faz referências a variáveis de ambiente da aplicação, definidas no arquivo oculto */.env*

3. Estrutura do projeto

3.1. Diretórios

Todos projetos Laravel possuem a mesma estrutura de diretórios padrão:

/app → Maior parte do aplicativo. Contém os models, controllers, definições de rota, comandos e código de domínio PHP

/bootstrap → Contém os arquivos usados na inicialização do Laravel

/config → Contém os arquivos de configuração da aplicação

/database → Contém as *migrations* (operações nas models) e *seeds*.

/public → É o diretório que o servidor aponta. Contém o *index.php* que é o controlador frontal que inicia o processo de bootstrapping e roteia as solicitações adequadamente. Contém arquivos voltados ao cliente, como imagens, css, scripts ou downloads

/resources → Contém arquivos não PHP necessários para realização de outros scripts. Views, e arquivos opcionais como Sass/Less e javascript

/routes → Contém as definições de rotas, tanto para rotas HTTP quanto para "rotas de console", bem como comando *Artisan*.

/storage → Contém cache, logs e arquivos compilados

/tests → Contém testes de unidade e integração

/vendor → Onde composer instala suas dependências. É por padrão incluída no *.gitignore*

3.2. Arquivos "soltos"

.env -> variáveis de ambiente (Informações da aplicação, Banco de dados, Broadcast/session/queue, smtp). É por padrão incluído no *.gitignore*. Suas variáveis são acessadas, internamente na aplicação, pelo helper env().

.env.example -> Arquivo padrão que é duplicado para criar o .env

package.json -> Arquivo que lista os pacotes Javascript a serem instalados na aplicação ao executar *npm install* na raiz.

4. Banco de dados

O Laravel possibilita o usuário a criar, consultar e manipular banco de dados de forma ágil e fácil.

4.1. Models

As models são classes que modelam uma entidade e a torna acessível para a aplicação de uma forma dinâmica. Podemos dizer que são usadas como uma interface entre o banco de dados e a aplicação.

O seguinte comando, executado na raiz do projeto, criará uma model de nome ***nome_model*** em *App/nome_model.php*.

```
php artisan make:model nome_model
```

Acesse-o e realize as modificações conforme os campos de sua entidade. Os atributos definidos no campo **$fillable**, serão de atribuição em massa, ou seja, não existirá proteção no caso de um usuário passar um parâmetro inesperado por uma requisição HTTP que alterará uma coluna no banco de dados, por exemplo. Para contornar isso, existe o campo **$guarded** que garante que seus atributos sejam protegidos quanto a atribuição em massa. Por *default*, todos modelos *Eloquent* são protegidos.

O campo **$table** definirá o nome da model. Por default, assume o plural (em inglês) do nome do nome da classe.

Exemplo abaixo:

```php
// app/Pessoa.php

class Pessoa extends Model
{
    protected $fillable = [
        'nome',
        'cpf'
    ];

    protected $table = 'Pessoas';
}
```

	Deve ser usado **$fillable** ou **$guarded** - não os dois. Todos não assinalados como **$fillable** serão **$guarded** e vice-versa

4.2. Migrations

Uma migration é responsável por modelar, criar e deletar um banco de dados referente a uma model.

Criando uma migration:

```
php artisan make:migration nome_migration --create nome_model
```

Criará um arquivo default de migration em */database/migrations*.

```
public function up()
    {
        Schema::create('_log', function (Blueprint $table) {
            $table->increments('id');
            $table->string('MAC');
            $table->string('status');
            $table->string('error');
            $table->string('time');
            $table->timestamps();
        });
    }
```

Os campos **timestamps**() e **increments**() são padrões em Laravel.

Os campos comuns de *Blueprint* estão listados no apêndice I.

O construtor de esquemas do Laravel permite apenas comandos do padrão ANSI SQL. Para executar demais comandos, use *DB::statement()*.

Executando as migrações:

```
php artisan migrate
```

Executará as migrações pendentes, criando, editando ou dropando as tabelas no banco de dados configurado em */.env*.

Demais opções de comandos de migração no Apêndice II.

Não delete uma migration sem antes executar *php artisan migrate:reset*, caso contrário ocorrerá inconsistência em seu banco de dados!

4.3. Testando o banco

4.3.1. Ferramenta Tinker

Tinker é um REPL (read-eval-print-loop) para interação direta com a aplicação. O iniciamos com seguinte comando:

```
php artisan tinker
```

Podemos criar um registro utilizando a model *Pessoa* da seguinte forma:

```
>> App\Pessoa::create(['nome' => 'Gabriel', 'sobrenome' =>
'Melo' , 'cpf' => '012.345.678-99']);
```

Podemos retornar todas os registros da model no BD com:

```
>> App\Pessoa::all();
```

Demais métodos podem ser consultados na seção do *Eloquent*, o ORM usado pelo Laravel.

4.3 Relacionamento entre Models:

Podemos facilmente relacionarmos dois models por FOREIGN KEY em Laravel.

Suponha uma nova model ***Telefone*** que terá um relacionamento *muitos-para-um* com ***Pessoa***. Uma pessoa pode conter vários telefones, mas cada telefone pertence a somente uma pessoa.

```
// /database/migrations/2018000000002_create_telefone_table.php
public function up() {
        Schema::create('Telefones', function (Blueprint $table) {
            $table->increments('id');
            $table->string('DDD');
            $table->string('numero');
            $table->integer('id_pessoa')->unsigned();
            $table->foreign('id_pessoa')->references('id')-
>on('Pessoas')->onDelete('cascade');
            $table->timestamps();
        });
}
```

Tenha certeza de que os tipos da *foreign key* e a referência são iguais. Certifique, também que a referência é uma *primary key*.

Criada a model e sua migration, podemos editar a classe *Pessoa* para acessarmos os telefones referentes a cada instância:

```
// /app/Pessoa.php
public function telefone() {
        return $this->hasMany(Telefone::class, 'id_pessoa');
}
```

O mesmo pode ser feito na classe *Telefone* para acessar os dados do dono do telefone:

```
// /app/Telefone.php
public function pessoa()  {
        return $this->belongsTo(Pessoa::class,'id_pessoa');
}
```

Pelo *tinker*, podemos testar o relacionamento:

```
>> App\Pessoa::find(1)->telefone;
```

O método find() retorna do registro da tabela, buscando pela propriedade $primaryKey. No nosso caso, o atributo *id*.

4.4. Seeding

Os seeds são alimentos para uma tabela do banco de dados e, em Laravel, são localizados em */database/seeds*. Existe, por padrão, a classe DatabaseSeeder que possui o método run() que é chamado quando chamamos o seeder.

Podemos executar um seeder junto com uma migração:

```
php artisan migrate --seed
```

Ou o executamos independentemente:

```
php artisan db:seed // Seed para todas as classes

php artisan db:seed --class CLASSE // Seed para uma classe
```

Para criarmos um seeder:

```
php artisan make:seeder SUA_TABELA_SEEDER
```

Após criado, vamos adicionar o seeder à classe DatabaseSeeder para que possa ser executado:

```
public function run() {
        $this->call(SUA_TABELA_SEEDER::class);
    }
```

Agora editaremos nosso arquivo *SUA_TABELA_SEEDER.php* para executar uma simples inserção com a Facade DB:

FAZENDO...

Desta forma, teremos um registro em nossa tabela, porém para que o seeder possa ser realmente funcional, devemos possuir mais registros. Conseguimos isso de forma automatizada com uma Model Factory.

Model Factories PAG 172

FAZENDO...

Facade DB PAG 176

FAZENDO...

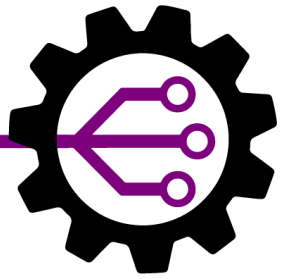
Eloquent PAG 189

FAZENDO...

Artisan

FAZENDO...

5. Templates com Blade



Inspirado na **Razor** da plataforma .NET, o **Blade** é a elegante -*e limpa*- engine de templates do Laravel. O componente fornece rapidez em atividades corriqueiras e também acessibilidade e facilidade em requisitos complexos, como herança aninhada e recursão.

5.1 Ecoando Dados

As chaves duplas **{{** e **}}** são usadas para delimitar as seções de PHP que serão ecoadas.

```
{{ $variavel }}
```

É o equivalente a:

```
<?php htmlentities($variavel) ?>
```

5.2 Estruturas de Controle

As tags em Blade são prefixadas com a diretiva @. Suas estruturas de controle têm aparência mais *clean* quando comparadas com o PHP puro.

5.2.1 Condicionais

@if, @endif

Compilado para:

```
<?php if ($condicao): ?>
```

O mesmo serve para @else e @elseif.

@unless, @endunless

Mesmo que:

```
@if (!$variavel)
```

or

Helper que substitui o `isset()`. Confere se variável foi declarada **ou** retorna um valor padrão definido. Como em:

```
{{ $variavel or "default" }}
```

5.2.2 Repetições

@for, @endfor

For tradicional.

@foreach, @endforeach

For iterando itens/objetos.

@forelse, @endforelse

For com um Fallback adicional ao objeto de iteração se esvaziar.

5.3 Herança

O Laravel possui diretivas que definem seções que podem ser estendidas e reutilizadas por templates-filhos:

@yield

Define uma seção com nome específico. Pode ter um valor default, caso a seção não seja estendida. O primeiro parâmetro de uma seção é sempre seu nome. Veja os exemplos:

```
<title> @yield('titulo', 'Meu título') </title>
```

Neste caso, a tag `<title>` conterá *"Meu título"* caso o template-pai não seja extendido, ou conterá o valor atribuído à seção *'titulo'* no template que o estender.

Observe outra situação:

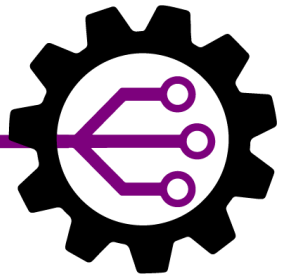
```
<p> @yield('conteudo') </p>
```

Neste caso, a tag `<p>` só conterá valores caso seja estendida.

Podemos controlar se uma seção possui uma sobrescrita (ou pegar seu valor) de um template-filho através de:

```
@if (trim($__env->yieldContent('conteudo')))
```

@section, @show



Seção do template-pai que pode definir um bloco inteiro como fallback padrão. Seu conteúdo pode ser sobrescrito por templates-filhos (como no caso da diretiva @yield), mas também pode ser disponível pela diretiva @parent nos filhos.

Nos filhos as seções são iniciadas também por @section, porém são finalizadas por @endsection.

@include

Seção que permite abrir uma view a partir de outra. É possível passar dados de maneira explícita, através do segundo parâmetro da diretiva @include, mas também podemos referenciar variáveis do arquivo que incluiu essa view.
Veja os exemplos:

```
@include('botao-login', ['texto' => 'Faça seu login'])
```

Neste caso, a diretiva incluirá a view localizada em *resources/views/botão-login.blade.php*, passando a variável texto como parâmetro. Essa view poderia, de uma forma bem simples, ser desta forma:

```
<a class="button"> {{ $text }} </a>
```

5.4 View composers

FAZENDO...

6. Componentes Front-end com mix

O Laravel conta com uma poderosa ferramenta *Javascript*, chamada **mix,** para compilar assets, como pré-processadores Sass ou LESS, copiar, concatenar e minificar arquivos e muito mais. O **mix** substitui o **gulp**.

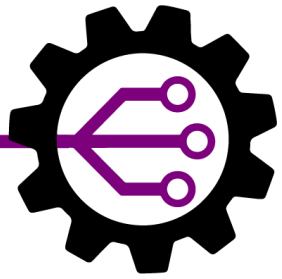
6.1. Instalação

Caso seu *package.json* não inclua a instalação do **mix** ao executar no diretório raiz do projeto:

```
npm install
```

Instale-o, manualmente, no diretório raiz do projeto com:

```
npm install laravel-mix
```



6.2. Arquivos e pastas Default

O arquivo Sass padrão é localizado em `resources/assets/sass/app.scss`, enquanto o arquivo JS padrão está em `resources/assets/js/app.js`. Os arquivos padrões gerados estarão, respectivamente, em `public/css/app.css` and `public/js/app.js`.

O arquivo base onde as tarefas são dispostas é o *webmpack.mix.js*.

6.3. Usando o mix

O mix é chamado por:

```
npm run dev
```

que executa todas as tarefas dispostas no arquivo base.

Para manter a ferramenta assistindo mudanças em tempo real, utilize:

```
npm run watch
```

Tenha certeza de que sua biblioteca *node-sass* (que é um suporte Node.js para o compilador Sass *LibSass*) está atualizada.

```
npm rebuild node-sass
```

6.4. Métodos primários do Mix

Você pode compilar Sass:

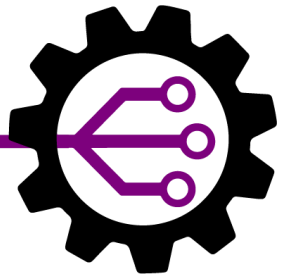
```
mix.sass('resources/assets/sass/app.scss', 'public/css');
```

Ou LESS:

```
mix.less('resources/assets/less/app.less', 'public/css');
```

Combinar arquivos:

```
mix.combine([
    'public/css/vendor/jquery-ui-one-thing.css',
    'public/css/vendor/jquery-ui-another-thing.css'
], 'public/css/vendor.css');
```



Copiar arquivos ou diretórios:

```
mix.copy('node_modules/jquery-ui/some-theme-thing.css',
'public/css/some-jquery-ui-theme-thing.css');
mix.copy('node_modules/jquery-ui/css', 'public/css/jquery-
ui');
```

Diferentemente do que acontece no **Elixir** (ferramenta do **gulp**), os source maps não são criados por padrão. Para ativar sua criação, basta adicionar ao aquivo:

```
mix.sourceMaps();
```

6.5. Versionamento e o Helper `mix()`

Muitos desenvolvedores concatenam aos seus assets compilados algum *timestamp* ou um token único de forma a forçar com que o navegador o carregue e atualize os assets ao invés de servir cópias de seu cache.

O **mix** manipula isso através do uso do método **version**, que concatena um hash único ao nome de todos os arquivos compilados:

```
mix.sass('resources/assets/sass/app.scss', 'public/css')
   .version();
```

Ou, de uma forma que faça mais sentido, podemos utilizar o versionamento de arquivos somente quando nossa aplicação estiver em produção:

```
If (mix.config.production) {
      mix.version();
}
```

Após gerar os arquivos "versionados", você não saberá o nome exato dos arquivos modificados. Desta forma, você deverá usar outra facilidade do Laravel para referenciá-los: o helper **`mix()`**.

```
<link href="{{mix('css/app.css')}}" rel="stylesheet" type="text/css">
```

6.6. Adicionando arquivos externos manualmente

Caso seja necessário – ou mais prático – você pode adicionar arquivos externos diretamente para o diretório `/public`. O Laravel fornece um ***helper asset*** que nos auxilia na hora de referenciar estes arquivos, da seguinte forma:

```
<link href="{{asset('css/style.css')}}" rel="stylesheet" type="text/css">
```

O helper `asset()` faz a referência para o diretório */public*

7. Roteamento e Controladores

As rotas apontam para determinado código quando receber determinada requisição do usuário. Os controladores, vêm como uma forma inteligente de ligar as **models** com as **views**, como vimos na primeira seção deste material, e nos ajudarão muito na construção de aplicações enxutas e inteligentes.

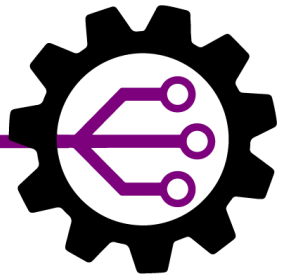
7.1. Rotas

Em Laravel, as rotas são definidas em */routes*. As rotas web, que têm retornos em html, são definidas em */routes/web.php*.

Um simples exemplo de rota:

```
Route::get('/', function(){
      return view(layouts.welcome);
});
```

Ao receber uma solicitação GET sem nenhum parâmetro, "/", (e.g. acessar *localhost/*, supondo que seu localhost seja definido na pasta */public* da aplicação), a rota chamará a view em resources/views/layouts/*welcome.blade.php*.



O mesmo funcionamento ocorre no exemplo abaixo, ao acessar sua aplicação com o parâmetro */contato*:

```
Route::get('/contato', function(){
      return view(layouts.contato);
});
```

A chamada de views diretamente das rotas é uma prática ruim. Veremos, mais adiante, que utilizar controllers para chamada das views é uma alternativa melhor e coerente com o padrão MVC.

7.1.1 Verbos de rota

Route::get(), Route::post()

Recebe e trata as requisições padrões do protocolo HTTP GET/POST.

Route::put(), Route::delete(), Route::any(), Route::match()

Recebe e trata demais requisições HTTP, geralmente usadas em APIs REST.

7.1.2. Parâmetros de rotas

Rotas que tiverem segmento(s) da estrutura URL variável(is) podem ser acessadas por:

```
Route::get('/post/{id}', function($id){
      //
});
```

Desta forma, a closure receberá como parâmetro a próprio valor de id na variável $id.

Note que os parâmetros da closure e os segmentos da URL não devem possuir a mesma nomenclatura. A relação entre eles é dada pela ordem que são dispostos.

7.1.3. Restrições com regex

Podemos restringir os parâmetros aceitos por uma rota utilizando expressões regulares (regexes), como no exemplo abaixo:

```
Route::get('/post/{id}', function($id){
      //
})->where('id', '[0-9]+');
```

7.1.4. Nome de rotas

Podemos definir nome às nossas rotas de forma a acessá-las internamente em nossa aplicação de forma mais fácil.

```
Route::get('/', function(){
      //
})->name('welcome');
```

7.1.5. Helper route()

De forma alternativa, podemos utilizar o helper url() para referenciar uma rota sem fazê-la explicitamente ou usando um nome.

```
<a href="{{url('/')}}">
```

7.1.6. Helper url()

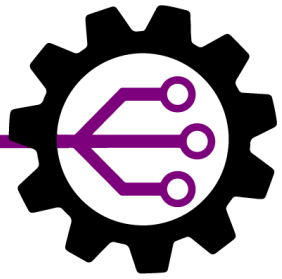
De forma alternativa, podemos utilizar o helper url() para referenciar uma rota sem fazê-la explicitamente ou usando um nome.

```
<a href="{{url('/')}}">
```

7.1.7. Grupo de rotas

Podemos agrupar algumas rotas que compatilhem alguma característica, como mesmo prefixo, requisitos de autenticação e namespaces de controladores.

7.1.7.1. Middleware



Camada usada, dentre alguns objetivos, para autenticar usuários e impedir acesso de usuários não autorizados em determinadas partes da aplicação. Por exemplo:

```
Route::group(['middleware' => 'auth'], function()
{
      Route::get('member', function() {
            Return view('layout.member');
      });

      Route::get('admin', function() {
            Return view('layout.admin');
      });
});
```

Nesta situação, o usuário só receberá a view esperada – *member* ou *admin* – caso seja autenticado.

7.1.7.2. Prefixos de caminho

Para simplificar a estruturação e legibilidade do código, podemos usar o agrupamento por prefixos de caminho, como exemplificado abaixo:

```
Route::group(['prefix' => 'post'], function() {
      Route::get('/', function() {
            Return view('layout.posts');
      });

      Route::get('/{id}', function($id) {
            Return view('layout.postId')
                  ->with('id', $id);
      });
});
```

7.1.8. Chamando views com parâmetros

Ainda utilizando o exemplo a cima, podemos notar que utilizamos o método adicional *with()*. Esse método injeta variáveis às views.

7.2. Controladores

Os controladores residem, por padrão, em *app/Http/Controller*. Para criar um novo controler, utilizamos, mais uma vez, o *Artisan*:

```
php artisan make:Controller nome_controller
```

Os controladores são chamados nas rotas da seguinte forma:

```
Route::get('/home', 'HomeController@index')->name('home');
```

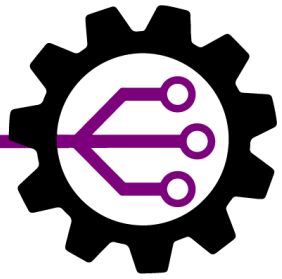
Este exemplo direcionará, pelo namespace padrão, a aplicação quando receber a rota para execução do método index() no arquivo *app/Http/Controller/HomeController.php*.

7.2.2. Redirecionamentos

7.2.2.1. redirect()->with()

As variáveis vão para Session
Conferimos com Session::has('msg')
e pegamos com Session::get('msg') que retorna a variável ou array.

FAZENDO...



Apêndice 1

1. Métodos de campos de Blueprint

integer(*nome_coluna*), **tinyInteger**(*nome_coluna*), **smallInteger**(*nome_coluna*), **mediumInteger**(*nome_coluna*) e **bigInteger**(*nome_coluna*)

Adiciona uma colina do tipo INTEGER ou uma de suas variações.

string(*nome_coluna, tamanho OPCIONAL)*

Adiciona uma coluna do tipo VARCHAR.

binary(*nome_coluna*)

Adiciona uma coluna do tipo BLOB.

boolean(*nome_coluna)*

Adiciona uma coluna do tipo BOOLEAN (TINYINT(1) em MySQL).

char(*nome_coluna, tamanho)*

Adiciona uma coluna do tipo CHAR.

datetime(*nome_coluna)*

Adiciona uma coluna do tipo DATETIME.

decimal(*nome_coluna, precisao, escala)*

Adiciona uma coluna do tipo DECIMAL.

double(*nome_coluna, total_de_digitos, dígitos_apos_virgula)*

Adiciona uma coluna do tipo DOUBLE.

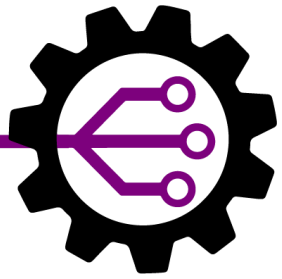
enum(*nome_coluna, [opção_1, opção_2,...])*

Adiciona uma coluna do tipo ENUM.

float(*nome_coluna)*

Adiciona uma coluna do tipo FLOAT (DOUBLE em MySQL).

json(*nome_coluna)*



Adiciona uma coluna do tipo JSON.

text`(nome_coluna)`, ***mediumText***`(nome_coluna)` e ***longText***`(nome_coluna)`

Adiciona uma coluna do tipo TEXT ou suas variações.

time`(nome_coluna)`

Adiciona uma coluna do tipo TIME.

timestramp`(nome_coluna)`

Adiciona uma coluna do tipo TIMESTRAMP.

uuid`(nome_coluna)`

Adiciona uma coluna do tipo UUID (CHAR(36) em MySQL).

2. Métodos de propriedades adicionais

nullable()

Permite valores NULL na coluna.

default(*'valor_padrao'*)

Define um valor padrão para coluna caso nenhum valor seja fornecido.

unsigned()

Define coluna de inteiros sem sinal.

first()

Insere a coluna em primeiro lugar na ordem das colunas.

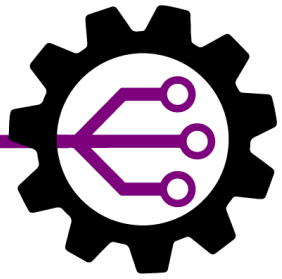
after(*nome_coluna*)

Insere a coluna após outra coluna na ordem das colunas.

unique()

Adiciona um índice UNIQUE.

primary()



Adiciona um índice PRIMARY KEY. Usa-se **primary**([*'chave_1','chave_2'*]) para chaves compostas.

index()

Adiciona um índice básico.

foreign(*nome_coluna*)->**references**(*nome_coluna*_externa)->
on(*nome_tabela*);

Estabelece uma FOREIGN KEY. Podem ser usadas, também, restrições de chave externa como **onDelete**() e **onUpdate**().

Os métodos **first**() e **after**() só se aplicam em MySQL.

Apêndice 2

1. Métodos alternativos de migração

php artisan migrate:refresh

FAZENDO...

Referências

1. Desenvolvendo com Laravel: Um framework para desenvolvimento de aplicativos PHP modernos, Matt Stauffer;
2. PHP: Programando com orientação a objetos, Pablo Dall'Oglio;
3. Fundamentos de HTML5 e CSS3, Maurício Samy Silva;
4. Documentação oficial Laravel[3];

[3] *https://laravel.com/docs/5.6*