UTFPR

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DIO CEZAR GO

LARAVEL

Parte 1

O QUE É?

O QUE É O LARAVEL?

- o *Laravel* é um *framework* escrito em PHP;
- foi construído com base nos componentes do symfony2;
  - https://symfony.com/
  - o *symfony* é também um *framework* e define um conjuntos de componentes reutilizáveis em PHP;
- como todo *framework*, fornece serviços e bibliotecas para realizar a interação entre as requisições web e outros serviços;
  - banco de dados, por exemplo;

INSTALAÇÃO

INSTALAÇÃO

- deve-se analisar se o servidor possui os seguintes requisitos:
  - PHP >= 7.0.0
  - *OpenSSL PHP Extension*
  - *PDO PHP Extension*
  - *Mbstring PHP Extension*
  - *Tokenizer PHP Extension*
  - *XML PHP Extension*
- o *Laravel* utiliza o *Composer* para gerenciar suas dependências;
  - tenha certeza que ele está instalado e configurado em sua máquina;

INSTALAÇÃO

- existem 2 formas simples e básicas para a sua instalação;
- a primeira utiliza o *Composer* para instalar um binário global de controle de instalações do *Laravel*.

```
composer global require "laravel/installer"
```

- com isso, é possível executar comandos como:

```
laravel new blog
```

INSTALAÇÃO

- a segunda forma, utiliza o próprio *Composer* para a instalação de um novo projeto, podemos então utilizar o seguinte comando:

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

INSTALAÇÃO

- se o PHP estiver instalado em seu ambiente de desenvolvimento, você pode o Artisan para servir a sua aplicação, e muito mais...;

```
php artisan serve
```

CONFIGURAÇÕES

CONFIGURAÇÕES DO PROJETO

- o *Laravel* trabalha com um diretório específico para servir os documentos públicos;
- pode-se notar que existe uma pasta */public*
- a recomendação é que seu servidor *web* aponte para essa pasta e não para a pasta raiz da aplicação;
- sem isso, toda vez que você acessar sua aplicação deverá fazer isso em */public*
- existem alternativas para trazer a pasta “public” para a raiz, através de simples alterações na estrutura do *framework*;

CONFIGURAÇÕES DO PROJETO

- todos os arquivos de configuração estão na pasta *config*;
- cada uma das opções é documentada com comentários;
- sinta-se à vontade para navegar entre os arquivos e analisar qual configuração você precisará alterar;

DIRETÓRIOS E PERMISSÕES

- depois de instalar o Laravel, talvez seja necessário configurar algumas permissões (ambientes Linux?)
- os diretórios *storage* e *bootstrap/cache* precisam ter a permissão de escrita por seu servidor *web*;

APPLICATION KEY

- a próxima configuração importante é setar sua aplicação com uma chave aleatórial
- se você instalou sua aplicação pelo composer, essa chave provavelmente já foi gerada;
- essa é uma chave utilizada por vários componentes do sistema para a geração de criptografias específicas da sua aplicação;
- a chave pode ser setada no arquivo *.env* (ambiente)
- se quiser fazer isso automaticamente: `php artisan key:generate`

CONFIGURAÇÕES ADICIONAIS

- quase tudo está ok, e você está pronto para começar o desenvolvimento;
- entretanto, você deve revisar o arquivo *config/app.php* que contém uma série de opções como *timezone* e *locale*;
- e é nesse arquivo que você poderá também configurar coisas como:
  - cache;
  - banco de dados;
  - sessão;
  - envio de emails;

CONFIGURANDO O AMBIENTE

- note que, na pasta raiz da aplicação, temos um arquivo `.env`
- este é um arquivo simples com chaves e valores;
- estes valores são utilizados nos arquivos de configuração;
  - exemplo: `'host' => env('DB_HOST', '127.0.0.1')`
  - note que o Laravel tenta achar a variável no ambiente `DB_HOST` ou então o outro valor;
- a grande sacada é deixar vários ambientes já configurados;
- imagine que você terá um ambiente de desenvolvimento e um de produção;
- crie dois `.env` dê nomes e valores diferentes a eles e altere apenas em seu `config/app.php` o ambiente *default*;
- mais detalhes: https://goo.gl/ogfXjr

ESTRUTURA DE DIRETÓRIOS

A ESTRUTURA DE DIRETÓRIOS

- as coisas mudam um pouco entre versões diferentes;
- vamos estudar e compreender a estrutura da versão 5.5;
- essencialmente, altera-se apenas a localização dos arquivos, mas os conceitos e suas aplicações são os mesmos;

A ESTRUTURA DE DIRETÓRIOS

- app → contém o core da sua aplicação;
- bootstrap → contém o *app.php* que inicializa o *framework*;
- config → contém os arquivos de configuração;
- database → é um diretório específico para manter os database *migrations* e *seeds;*
- public → é o diretório que contém o *index.php* que é o arquivo de entrada para todos os requests da sua aplicação, também possui *assests* como imagens, javascripts e css;

A ESTRUTURA DE DIRETÓRIOS

- resources → guarda as *views* da aplicação, além disso, possui os assets originais para guardar arquivos SASS ou JavaScripts não minificados;
- routes → contém todas as definições de rotas da sua aplicação;
- storage → contém os arquivos compilados do blade, sessões baseadas em arquivos, arquivos de cache e outros arquivos gerados pelo *framework*;
- tests → é o diretório que abriga seus testes automatizados; pode-se criar testes com o PHPUnit por exemplo e configurá-los neste diretório;
- vendor → é o diretório que contém as dependências do *Composer*;
- mais sobre a estrutura: https://goo.gl/6C1sBf

ROTAS

ROTAS

- as rotas são definidas nas pastas routes;
- em `web.php` temos as rotas da aplicação;
- as rotas utilizam o conceito de *closure;*
  - *closure* é a capacidade de se passar por parâmetro uma função;
- com as rotas você pode escrever um comando diretamente dentro da função, ou...
- chamar uma *controller*;
- mais sobre as rotas : https://goo.gl/EoVUPo

ROTAS

ROTEAMENTO COM CLOSURE

```
Route::get('/ola', function () {
    return “Olá”
});

Route::get('/ola/alo', function () {
    return “Alo”
});

Route::get('user/{id}', function ($id) {
    return 'User '. $id;
});
```

ROTAS

ROTEAMENTO CONTROLLER

```
Route::get('/user', 'UsersController@index');
Route::get('/user/id', 'UsersController@index');

// na controller

public function index($id){
      … $id …
}
```

ROTAS

UTILIZAÇÃO DE PREFIXOS

```
Route::prefix('admin')->group(function () {
    Route::get('users', function () {
        // Matches The "/admin/users" URL
    });
});
```

MIDDLEWARES

MIDDLEWARE

- os *middlewares* são mecanismos para filtros dos *requests* em toda a sua aplicação;
- por exemplo, o *Laravel* inclui um *middleware* que verifica se o usuário da aplicação está autenticado;
- se o usuário não está autenticado, *middleware* redireciona o usuário para a tela de login;
- se o usuário está autenticado, permite-se o *request* que seja efetuado;
- pode-se criar outros tipos de *middlewares* de acordo com as necessidades da aplicação;

MIDDLEWARE

EXEMPLO MIDDLEWARE LOGIN (SIMPLIFICADO)

```
namespace App\Http\Middleware;

class Authenticate {
      public function handle($request, Closure $next)  {
            if ($this->auth->guest()){
                  return redirect()->guest('/admin/auth/login');
            }
            return $next($request);
      }
}
```

CONTROLLERS

CONTROLLERS

- as *controllers* ficam em `app/Http/Controllers`
- utiliza o namespace que identifica a localização das *Controllers*;
    - `namespace App\Http\Controllers;`
- com `use` importa-se a *model* localizada em `App\User`;
- com `use` importa-se a *controller base* localizada em `App\Http\Controllers\Controller`;
- a classe herda a *controller* base;
- cria-se uma função *show* que retorna uma view;
- passa como parâmetro um *array* que em user recebe um acesso ao *Eloquent* de busca por um único registro com um id passado por parâmetro;

CONTROLLERS

EXEMPLO CONTROLLER

```
<?php

namespace App\Http\Controllers;

use App\User;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    public function show($id){
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

CONTROLLERS

ROTA PARA O CONTROLLER

```
Route::get('user/{id}', 'UserController@show');
```

HTTP REQUESTS

HTTP REQUESTS

- para obter os dados do *Request*, precisaremos utilizar a classe: `Illuminate\Http\Request;`
- dessa forma, teremos acesso dentro da *controller* sobre os dados de POST que vierem do *Front-End*;
- mais informações sobre os HTTP requests: https://goo.gl/8HZwGq

HTTP REQUESTS

EXEMPLO UTILIZAÇÃO

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller{
    public function store(Request $request)    {
        $name = $request->input('name');
    }
}
```

HTTP REQUESTS

- se o seu *controller* também receber parâmetros (provavelmente das rotas)

```
Route::put('user/{id}', 'UserController@update');
```

- então, pode-se utilizar os parâmetros depois do *request* como mostra o próximo exemplo;

HTTP REQUESTS

EXEMPLO UTILIZAÇÃO COM PARÂMETROS

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller{
    public function update(Request $request, $id){
        //
    }
}
```

VIEWS

VIEWS

- as *views* contém o HTML que será servido pela aplicação;
- elas estão localizadas em `resources/views`
- utiliza-se a sintaxe *Blade* para que basicamente faz o mesmo que que qualquer template para substituição de “tags” específicas HTML;
- mais detalhes sobre as views podem ser vistas em: https://goo.gl/uJohbr

VIEWS

EXEMPLO VIEW

```
<!-- View stored in resources/views/greeting.blade.php -->

<html>
    <body>
        <h1>Hello, {{ $name }}</h1>
    </body>
</html>

// Rota que chama a view

Route::get('/', function () {
    return view('greeting', ['name' => 'James']);
});
```

VIEWS

- apenas pelo fato de estar em `resources/views/greeting.blade.php` podemos retorná-la utilizando a *helper view*;
- os *helpers* são funções globais disponibilizadas em qualquer parte do código;
- obviamente iremos retornar uma view dentro de uma *controller*;
- você poderá organizar as *views* em pastas e utilizar o caractere “.” para referenciar sua *view*;
- exemplo:
  - `resources/views/admin/profile.blade.php → return view('admin.profile', $data);`

VIEWS

EXEMPLO CHAMADA DIRETA DA VIEW EM UMA ROTA

```
class CityController extends AdminBaseController {
      public function index(){
            $data = array(‘name’ => ‘Cornélio Procópio’);
            return $this->view('admin.city.index', $data);
      }
      public function create(){
            return $this->view('admin.city.save');
      }
}
```

BLADE

BLADE

- e como todo sistema de templates temos comandos específicos do Blade, como:
- `@extends('admin.layout.layout')` → importa um *layout* existente;
- `@section('content')` → define uma seção com o nome content;
- `@section('title', 'Minha Página')` → define uma seção e valor;
- `@stop` → encerra uma seção;
- `@yield('title')` → imprime o conteúdo de uma seção definida;
- mais informações sobre o blade podem ser vistas em: https://goo.gl/C2MqQc

BLADE

MASTER.BLADE.PHP

```
@section('title', 'Minha Página')
<!DOCTYPE html>
<html lang="en">
<head>
      <meta charset="utf-8">
</head>
<body>
      <div class="container">
            <div class="panel panel-default">
                  <div class="panel-body">
                        @yield('content')
                  </div>
            </div>
      </div>
</body>
</html>
```

BLADE

ROLES.BLADE.PHP

```
@extends('master')
@section('content')
      <h2>Teste de Conteúdo Interno</h2>
      <p>Imprimindo o variável @yield('title')</p>
@stop
```

VALIDAÇÕES

VALIDAÇÃO

- como vimos, validar os dados em *JavaScript* pode ser interessante e performático;
- mas... sempre temos que pensar no lado servidor.
  - e se o seu JavaScript for burlado? Bem fácil com o Google Chrome Dev Tools;
- por isso o *Laravel* implementa um recurso bastante interessante de validação de dados no lado servidor;
- mais detalhes sobre o validation: https://goo.gl/SDvVzR

VALIDAÇÃO

- basta estar com o objeto `$request` em mãos;
- passamos um `array` com parâmetros específicos para o método `validate`;
- se a validação falhar, um `response` automático será gerado chamando novamente a *view* que enviou os dados, mas agora com um objeto de erros que pode ser tratado e exibido para o usuário;
- se a validação estiver ok, então o fluxo de execução continua normalmente;
- opcionalmente, pode-se passar uma outro array com mensagens customizadas a serem exibidas, caso contrário as mensagens default de erros (de acordo com a linguagem do sistema) será exibida;

VALIDAÇÃO

VALIDAÇÃO NA CONTROLLER

```
class PostController extends Controller{
    public function create(){
        return view('post.create');
    }
    public function store(Request $request){
         $validatedData = $request->validate([
                'title'=> 'required|unique:posts|max:255',
                'body' => 'required'
           ],[
                'title.require' => 'Um título deve ser informado',
                'title.unique'  => 'Já existe um post com este nome',
                'title.max'     => 'No máximo 255 caracteres',
                'body.required' => 'Informe um conteúdo'
           ]);
         // The blog post is valid..
    }
}
```

VALIDAÇÃO

EXIBINDO OS ERROS NA VIEW

```
<h1>Create Post</h1>

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

THAT'S ALL FOLKS

MATERIAIS COMPLEMENTARES

- https://pt.slideshare.net/BrianFeaver/laravel-5-49041229
- https://laracasts.com/series/laravel-from-scratch-2017