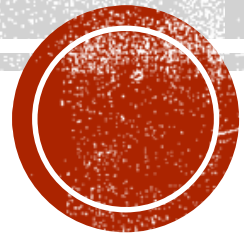
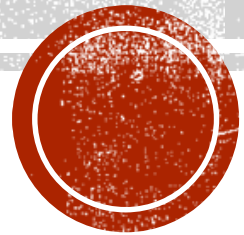
DESENVOLVIMENTO DE SISTEMAS WEB COM O FRAMEWORK LARAVEL

Thyago Maia Tavares de Farias



O QUE É UM FRAMEWORK?



FRAMEWORK

- Possui várias definições:
  - “Um conjunto de classe que encapsula uma abstração de projeto para a solução de uma família de problemas relacionados”;
  - “Um conjunto de objetos que colaboram para realizar um conjunto de responsabilidades para um domínio de subsistema de aplicativos”;
  - “Define um conjunto de classes abstratas e a forma como os objetos dessas classes colaboram”;
  - “Um conjunto extensível de classes orientadas a objetos que são integradas para executar conjuntos bem definidos de comportamento computacional”;
  - “Uma coleção abstraída de classes, interfaces e padrões dedicados a resolver uma classe de problemas através de uma arquitetura flexível e extensível”;

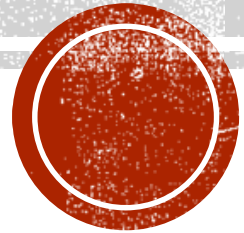
FRAMEWORK

- Observe que um framework é uma aplicação "quase" completa, mas com "pedaços" faltando:
  - Ao receber um framework, seu trabalho consiste em prover os "pedaços" que são específicos para sua aplicação;
  - Um framework provê funcionalidades genéricas, mas pode atingir funcionalidades específicas, por configuração, durante a programação de uma aplicação;

FRAMEWORK

- Vantagens:
  - Maior facilidade para detecção de erros;
  - Eficiência na resolução de problemas;
  - Otimização de recursos;
  - Concentração na abstração de solução de problemas;
  - Modelados com vários padrões de projeto;

O FRAMEWORK LARAVEL

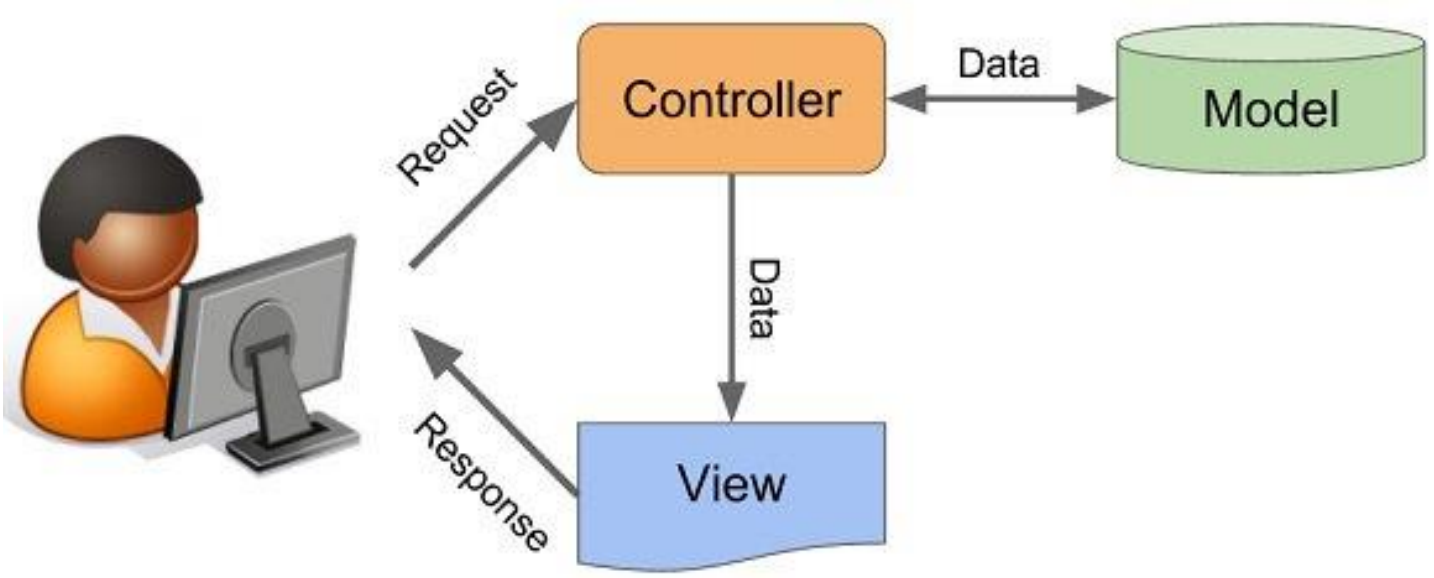


LARAVEL

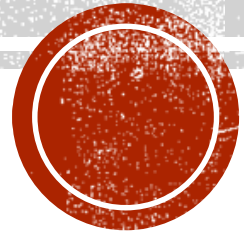
- Framework PHP open-source criado por Taylor Otwell;
- Objetiva o auxílio no desenvolvimento de aplicações Web baseados no padrão de projeto arquitetural MVC;
- Se tornou recentemente um dos frameworks PHP mais populares, ao lado do Symfony, Zend, CodeIgniter, entre outros;
- Hospedado no GitHub e licenciado nos termos da licença MIT;

LARAVEL

- O modelo MVC:



O AMBIENTE DE DESENVOLVIMENTO LARAVEL



AMBIENTE DE DESENVOLVIMENTO LARAVEL

- É possível configurar um ambiente de desenvolvimento Laravel a partir do Xampp. Para isso, será necessário a instalação do Composer;

AMBIENTE DE DESENVOLVIMENTO LARAVEL

- A partir de um prompt de comando, acesse a pasta C:\xampp\htdocs e execute o seguinte comando:

```
composer create-project laravel/laravel laravel "5.1.33"
```

- Será criada a pasta laravel na pasta htdocs do xampp, já com os arquivos de projeto Laravel. Para fazer com que o projeto fique disponível, no prompt de comando, digite:

cd laravel

php artisan serve

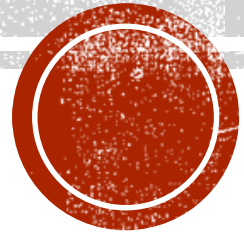
AMBIENTE DE DESENVOLVIMENTO LARAVEL

- Inicie o PHP e o MySQL no control panel do Xampp, clicando nos botões Start;

- Acesse a URL localhost:8000 e verifique se a página de boas vindas do Laravel será apresentada!

Laravel 5

REST



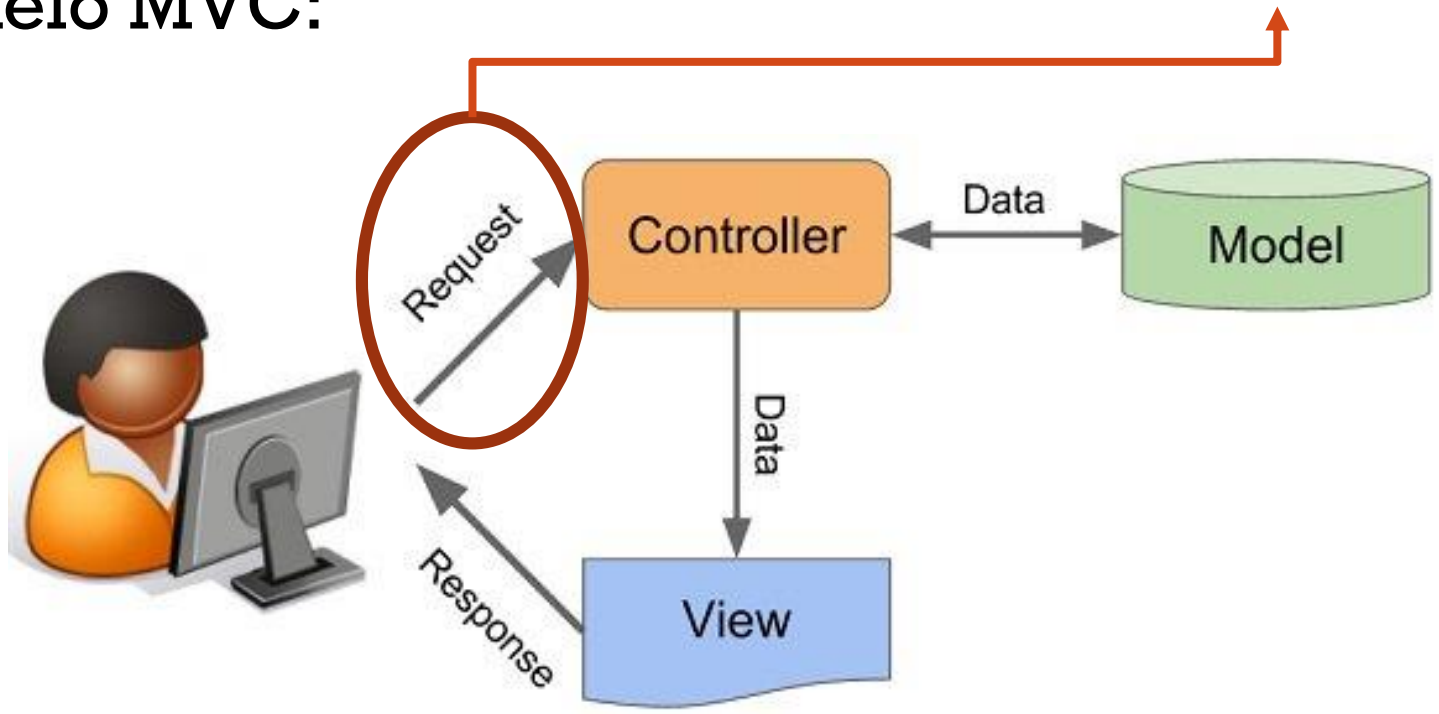
LARAVEL

- Seleciona controllers e/ou métodos de controller a partir do modelo REST:
  - Cada mensagem HTTP contém toda a informação necessária para compreensão de pedidos;
  - Utiliza as operações HTTP para a seleção de controllers e recursos de controller: POST, GET, PUT e DELETE;
  - Classifica operações de CRUD para a persistência de dados.
    - Ex.: Quando uma requisição HTTP do tipo DELETE é lançada para um controller, um método de exclusão de dados em banco poderá ser automaticamente executada;

LARAVEL

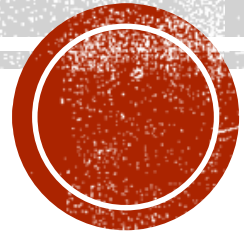
Utiliza um arquivo php de rotas para checar a operação HTTP requisitada e associá-la a um controller e/ou recurso de controller.

- O modelo MVC:



ROTAS LARAVEL

O arquivo Http/web.php



ROTAS LARAVEL

- A partir do arquivo web.php, é possível atribuir uma operação HTTP a um controller e/ou recurso de controller;
  - Ex.: Quando uma requisição HTTP do tipo get for lançada, execute o método da classe controller PrincipalController que apresenta a página de boas-vindas de uma dada aplicação web;

- Graças ao arquivo de rotas, não é necessário indicar na URL o recurso a ser acessado;
  - Ex.: Ao invés do link http://localhost:8000/pagina.php, poderíamos configurar uma rota para que o recurso fique acessível a partir do link http://localhost:8000/pagina
  - Assim, cada recurso da aplicação pode ser representado por um verbo;

ROTAS LARAVEL

- Atribuições de operações HTTP são realizadas a partir da classe Route e de seus métodos estáticos, que representam cada uma das operações HTTP existentes;

- Sintaxe básica para a definição de rota:

```
Route::<operação_HTTP>('/verbo',  function() {
      // O que será feito quando essa rota for acessada
});
```

ROTAS LARAVEL

- Exemplo: Abra o arquivo Http/web.php e crie a seguinte rota:

```
Route::get('/php-info',  function() {
      phpinfo();
});
```

- Abra o navegador, acesse http://localhost:8000/php-info e verifique se a página de informações sobre o servidor PHP será apresentada;

ROTAS LARAVEL

- Exemplo 2: Abra o arquivo Http/web.php e crie uma nova rota:

```
Route::get('/formulario',  function() {
      return '
      <form method="post" action="/contato">
      Nome: <input type="text" name="nome">
      Email: <input type="text" name="email">
      <input type="submit" value="Enviar">
      </form>';
});
```

ROTAS LARAVEL

- Exemplo 2: Abra o arquivo Http/web.php e crie uma nova rota:

```
Route::post('/contato',  function() {
      echo Request::input('nome');
      echo "<br/>";
      echo Request::input('email');
});
```

ROTAS LARAVEL

- Abra o navegador, acesse http://localhost:8000/formulario e verifique se o formulário criado no arquivo de rotas será apresentado. Digite suas informações de contato e os submeta, afim de verificar se a rota post será executada;

- Algum problema?

ROTAS LARAVEL

- Abra o navegador, acesse http://localhost:8000/formulario e verifique se o formulário criado no arquivo de rotas será apresentado. Digite suas informações de contato e os submeta, afim de verificar se a rota post será executada;

- Algum problema? SIM

  - Todo formulário Laravel precisa submeter um Token, chamado CSRF, para que possa enviar operações HTTP em aplicações Laravel (por questões de segurança!);

ROTAS LARAVEL

- Como só podemos inserir tais tokens em formulários implementados em Views (olha o MVC aí de novo!), por enquanto, vamos desligar esse recurso, comentando a linha de código que implementa esse recurso no arquivo app/Http/Kernel.php;

```
/**
 * The application's global HTTP middleware stack.
 *
 * @var array
 */
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \App\Http\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    //\App\Http\Middleware\VerifyCsrfToken::class,
];
```

ROTAS LARAVEL

- Abra o navegador, acesse http://localhost:8000/formulario e verifique se o formulário criado no arquivo de rotas será apresentado. Digite suas informações de contato e os submeta, afim de verificar se a rota post será executada;

ROTAS LARAVEL

- Métodos da classe Route disponíveis para rotas:

- Route::get($uri, $callback); - Utilizado de forma geral para navegar entre páginas;
- Route::post($uri, $callback); - Utilizado de forma geral para alterações no lado do servidor;
- Route::put($uri, $callback); - Utilizado de forma geral para atualizações de um recurso existente;
- Route::patch($uri, $callback); - Utilizado de forma geral para atualizações de um recurso existente;
- Route::delete($uri, $callback); - Utilizado de forma geral para remover um recurso existente;

ROTAS LARAVEL

- Também é possível definir vários verbos para uma única rota com o método match:

```
Route::match(['POST','GET'], '/formulario',  function() {
        if(Request::isMethod('post'))
                print_r(Request::all());
        else {
        return '
        <form method="post" action="/formulario">
        Nome: <input type="text" name="nome">
        Email: <input type="text" name="email">
        Mensagem: <textarea name="mensagem"></textarea>
        <input type="submit" value="Enviar">
        </form>';
        }
});
```

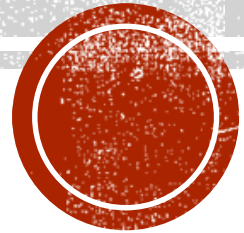
ROTAS LARAVEL

- Também é possível definir todos os verbos para uma única rota com o método any:

```
Route::any('/formulario',  function() {
        if(Request::isMethod('post'))
                print_r(Request::all());
        else {
        return '
        <form method="post" action="/formulario">
        Nome: <input type="text" name="nome">
        Email: <input type="text" name="email">
        Mensagem: <textarea name="mensagem"></textarea>
        <input type="submit" value="Enviar">
        </form>';
        }
});
```

ROTAS COM PARÂMETROS

O arquivo Http/web.php



ROTAS LARAVEL

- Até o momento, não passamos nenhum parâmetro nas nossas requisições (rotas). Mas em muitas aplicações, essa necessidade pode surgir;

- Para isso, a sintaxe na definição do verbo muda um pouco, onde será necessário explicitar os parâmetros na sua definição;

ROTAS LARAVEL

- Exemplo: Abra o arquivo Http/web.php e crie a seguinte rota:

```
Route::get('/contato/{id}',  function($id) {
      echo "Olá, o seu ID é {$id}";
});
```

- Abra o navegador, acesse http://localhost:8000/contato/25 e verifique se o ID 25 é apresentado no navegador. Se nenhum parâmetro for enviado, será gerada uma exceção;

ROTAS LARAVEL

- Já que nossa rota espera um id, vamos alterá-la para aceitar apenas números (atualmente ela também aceita strings!);

- Para isso, basta invocar o método where após a definição da rota, passando como parâmetro o nome do parâmetro e a expressão regular que impõe o uso de números;

ROTAS LARAVEL

- Exemplo: Abra o arquivo Http/web.php e crie a seguinte rota:

```
Route::get('/contato/{id}',  function($id) {
        echo "Olá, o seu ID é {$id}";
})->where('id','[0-9]+');
```

- Abra o navegador, acesse http://localhost:8000/contato/sport e verifique se o ID 'Sport' é apresentado no navegador. Se nenhum parâmetro for enviado, será gerada uma exceção;

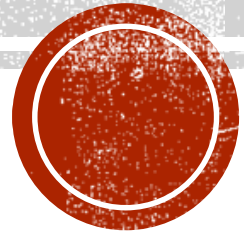
ROTAS LARAVEL

- Exemplo: Podemos passar mais de um parâmetro. Abra o arquivo Http/web.php e edite a seguinte rota:

```
Route::get('/contato/{id}/{nome}', function($id,$nome) {
        echo "Olá {$nome}, o seu ID é {$id}";
});
```

- Abra o navegador, acesse http://localhost:80000/contato/25/Thyago e verifique a saída apresentada no navegador. Se nenhum parâmetro for enviado, será gerada uma exceção;

ROTAS E CONTROLADORES



ROTAS LARAVEL

- E se minha aplicação tiver mil rotas????
  - Você não precisa criar todas as rotas no arquivo web.php!
  - Podemos mapear um controller no arquivo de rotas e, a partir dele, aplicar as rotas;
  - Para isso, em cada parâmetro passado na rota, devemos criar o parâmetro correspondente no método controller;

ROTAS LARAVEL

- O melhor? O Laravel cria o controller para você automaticamente!!!

- Para isso, vá no seu terminal, acesse a partir dele a pasta C:\xampp\htdocs\nomeProjeto e digite o comando:

```
php artisan make:controller Rotas
```

- Um novo controller com o nome Rotas será criado em app/Http/Controllers;

ROTAS LARAVEL

- Agora, só precisaremos de uma linha no arquivo de rotas:

```
Route::resource('rotas', 'Rotas');
```

- Abra o navegador, acesse http://localhost:8000/rotas. O método index do Controller Rotas será mapeado automaticamente na requisição;

ROTAS LARAVEL

- No controller app/Http/Controllers/Rotas.php, atualize o seguinte método:

```
public function index()     {
      return 'Olá, sou a rota padrão do controller!';
}
```

- Abra o navegador, acesse http://localhost:8000/rotas

ROTAS LARAVEL

- Também podemos criar o nome que desejarmos para nossas rotas. Exemplo: Logo abaixo do método index, crie o método:

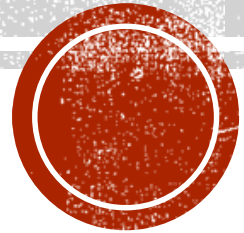
```
public function getRequisicao()     {
        return 'Olá, sou executado a partir de um GET';
}
```

- No arquivo de rotas, adicione a seguinte rota:

```
Route::get('/teste', 'Rotas@getRequisicao');
```

- Abra o navegador, acesse http://localhost:8000/teste

MIDDLEWARE HTTP



MIDDLEWARE HTTP

- Classes Laravel utilizadas para filtrar os dados de entrada dos nossos sistemas;

- Funciona como um filtro de requisição, que executa ações antes ou depois de uma requisição HTTP;

MIDDLEWARE HTTP

- Exemplo: Podemos inserir um token CSRF em formulários implementados em Views para que apenas nossos forms realizem requisições. Precisamos religar esse recurso, retirando o comentário da linha de código que implementa esse recurso no arquivo app/Http/Kernel.php;

```
/**
 * The application's global HTTP middleware stack.
 *
 * @var array
 */
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \App\Http\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    //\App\Http\Middleware\VerifyCsrfToken::class,
];
```

MIDDLEWARE HTTP

- Exemplo: Agora, vamos criar nossa primeira View! Vá para a pasta resources/views e crie o arquivo contato.blade.php. Edite-o com o seguinte código HTML:

```
<html>
        <body>
                <form method="post" action="enviar-contato">
                {{ csrf_field() }}
                <p>Nome: <input type="text" name="nome"></p>
                <p>Telefone: <input type="text" name="telefone"></p>
                <p><input type="submit" value="Enviar"></p>
                </form>
        </body>
</html>
```

MIDDLEWARE HTTP

- Exemplo: Agora, vamos criar nossa primeira View! Vá para a pasta resources/views e crie o arquivo contato.blade.php. Edite-o com o seguinte código HTML:

```
<html>
        <body>
                <form method="post" action="enviar-contato">
                {{ csrf_field() }}
                <p>Nome: <input type="text" name="nome"></p>
                <p>Telefone: <input type="text" name="telefone"></p>
                <p><input type="submit" value="enviar"></p>
                </form>
        </body>
</html>
```

Olha o middleware aqui gente!!!

MIDDLEWARE HTTP

- Exemplo: Agora, vamos criar uma rota para chamar nosso formulário de contato no arquivo de rotas:

```
Route::get('/contato', 'ContatoController@contato');

Route::post('/enviar-contato',
'ContatoController@enviarContato');
```

MIDDLEWARE HTTP

- Em seguida, criaremos o controller responsável por exibir e receber dados do formulário;

- Para isso, vá no seu terminal, acesse a partir dele a pasta C:\xampp\htdocs\nomeProjeto e digite o comando:

```
php artisan make:controller ContatoController
```

- Um novo controller com o nome ContatoController será criado em app/Http/Controllers;

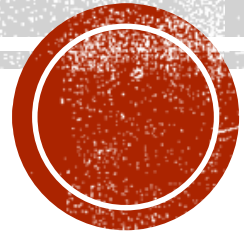
MIDDLEWARE HTTP

- Precisaremos criar os métodos contato() e enviarContato(Request $request) no controller em questão:

```
public function contato()     {
        return view('contato');
}
public function enviarContato(Request $request)      {
        return $request->all();
}
```

- Abra o navegador, acesse http://localhost:8000/contato

CONTROLADORAS (CONTROLLERS)



CONTROLLERS

- Até o momento, criamos uma controladora que responde a requisições HTTP do tipo GET e POST;

- Mas... E quando eu precisar responder outros tipos de requisição, como PUT ou DELETE?;

CONTROLLERS

- Exemplo: Vá no seu terminal, acesse a partir dele a pasta C:\xampp\htdocs\nomeProjeto e digite o comando:

```
php artisan make:controller BasicoController
```

- Um novo controller com o nome BasicoController será criado em app/Http/Controllers;

MIDDLEWARE CUSTOMIZADOS

- No controller BasicoController, adicione os seguintes métodos:

```
public function putBasico()     {
        return view('put');
}
public function put (Request $request)     {
        echo("Nome digitado: " . $request->input('nome'));
}
```

CONTROLLERS

- Em seguida, vamos atualizar nosso arquivo de rotas:

```
Route::get('/put', 'BasicoController@putBasico');
Route::put('/put', 'BasicoController@put');
```

CONTROLLERS

- Agora, começaremos a manipular a camada de visualização de dados. Crie a view put.blade.php na pasta resource/views:

```
<html>
<body>
        <form method="post" action="/put">
                {{ csrf_field() }}
                <input type="hidden" name="_method" value="PUT">
                <p>Nome: <input required type="text" name="nome"></p>
        </form>
</body>
</html>
```

CONTROLLERS

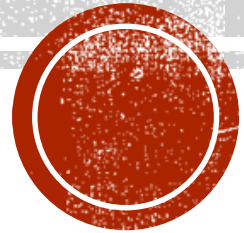
- Abra o navegador e acesse http://localhost:8000/put

CONTROLLERS

- Uma outra forma de alterar o tipo de requisição em um form HTML:

```
<html>
<body>
        <form method="post" action="/put">
                {{ csrf_field() }}
                {{ method_field('PUT') }}
                <p>Nome: <input required type="text" name="nome"></p>
        </form>
</body>
</html>
```

CONTROLLERS DE RECURSO



CONTROLLERS DE RECURSO

- Controllers também podem ser concebidas com o objetivo de abstrair funcionalidades CRUD (Criação, Leitura, Atualização e Exclusão);

- O Laravel pode não só criar tais tipos de classe automaticamente, como também as assinaturas de método necessárias e associar um tipo de requisição para cada método;

CONTROLLERS DE RECURSO

- Exemplo: Vá no seu terminal, acesse a partir dele a pasta C:\xampp\htdocs\laravel e digite o comando:

```
php artisan make:controller LivroController
```

- Um novo controller com o nome LivroController será criado em app/Http/Controllers;

CONTROLLERS DE RECURSO

- Verifique que o Laravel já criou as assinaturas de método de CRUD automaticamente:

- index()
- create()
- store(Request $request)
- show($id)
- edit($id)
- update(Request $request, $id)
- destroy($id)

CONTROLLERS DE RECURSO

- Outro poder de controladoras de recurso é a simplificação do registro de rotas. Continuando nosso exemplo, acesse o arquivo de rotas e registre a seguinte rota:

```
Route::resource('livro', 'LivroController');
```

CONTROLLERS DE RECURSO

- Com esse registro de rota, o Laravel automaticamente atribui as seguintes requisições HTTP e as rotas para cada método da controladora de recurso:

Tipo	URI	Ação
GET	/livro	index
GET	/livro/create	create
POST	/livro	store
GET	/livro/{id}	show
GET	/livro/{id}/edit	edit
PUT/PATCH	/livro/{id}	update
DELETE	/livro/{id}	destroy

CONTROLLERS DE RECURSO

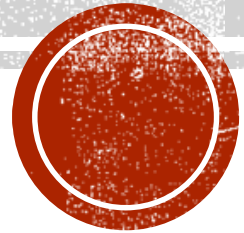
- Continuando o exemplo, no controller LivroController, edite os seguintes métodos:

```
public function index()     {
        return 'Olá, usuário!';
}
public function create ()     {
        return 'Aqui acessarei views para a inserção de dados!';
}
```

CONTROLLERS DE RECURSO

- Abra o navegador e acesse http://localhost:8000/livro e http://localhost:8000/livro/create para testar as rotas baseadas em métodos do nosso controller de recurso;

VIEWS



VIEWS

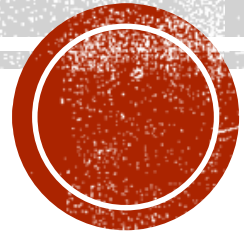
- As Views são módulos do Modelo MVC responsáveis pela exibição de dados para o usuário após um determinado processamento;

- O Blade é o motor de renderização que o Laravel adota (notaram o .blade inserido no nome dos arquivos de view?);

- O Blade permite uma sintaxe mais enxuta na criação de layouts, além de disponibilizar comandos para a geração de conteúdo dinâmico (geralmente começam por @);

RECURSOS DE LAYOUT

Recursos dinâmicos fornecidos pelo Blade



RECURSOS DE LAYOUT

- É possível utilizar PHP puro nas views para geração de conteúdo dinâmico. Mas isso é uma boa prática???
  - O ideal é só inserir scripts PHP nos controllers e models!
- O Blade fornece uma sintaxe e instruções que tornam a codificação de Views mais enxuta e organizada;

RECURSOS DE LAYOUT

- Exemplo: Acesse o arquivo de rotas e registre a seguinte rota:

```
Route::get('/exibirdados', function() {

    return view(‘exibirdados');
});
```

RECURSOS DE LAYOUT

- Crie na pasta /resources/views a view exibirdados.blade.php e acesse http://localhost:8000/exibirdados :

```
<h1>{{ $titulo ?? 'Título não encontrado' }}</h1>
<p>{{ $texto ?? 'Texto não encontrado' }}</p>
```

- If/else muito enxuto!

RECURSOS DE LAYOUT

- Exemplo: Agora, acesse o arquivo de rotas , atualize a rota exibirdados e acesse http://localhost:8000/exibirdados :

```
Route::get('/exibirdados', function() {
    return view('exibirdados', [
        'titulo' => 'Meu Blog',
        'texto'  => 'Sejam bem vindos!'
    ]);
});
```

RECURSOS DE LAYOUT

- O Blade também permite escapar dados de maneira simples;

- Escapar dados é muito importante por questões de segurança. Assim, evitamos que o usuário envie dados maliciosos a partir de técnicas como SQL injection e XSS;

RECURSOS DE LAYOUT

- Exemplo: Acesse o arquivo de rotas e registre a seguinte rota:

```
Route::get('/naoesc', function() {
    return view(‘naoesc‘, [
        ‘conteudo’ => ‘<h1>Sport</h1>’
    ]);
});
```

RECURSOS DE LAYOUT

- Crie na pasta /resources/views a view naoesc.blade.php e acesse http://localhost:8000/naoesc :

```
{{ $conteudo }}
```

- Com escape, as tags HTML que foram enviadas como parâmetros de view foram “escapadas”;

RECURSOS DE LAYOUT

- Agora edite a view naoesc.blade.php e acesse http://laravel.dev/naoesc :

```
{!! $conteudo !!}
```

- Sem escape, as tags HTML que foram enviadas como parâmetros de view foram interpretadas na aplicação;

RECURSOS DE LAYOUT

- O Blade também permite inserir estruturas de controle e repetição em views;

- Tais estruturas auxiliam na geração de conteúdo dinâmico a partir de uma sintaxe bastante enxuta;

RECURSOS DE LAYOUT

- Exemplo: Acesse o arquivo de rotas e registre a seguinte rota:

```
Route::get('/se', function() {
    return view(‘vazio‘, [
        ‘lista’ => []
    ]);
});
```

RECURSOS DE LAYOUT

- Agora edite a view vazio.blade.php e acesse http://localhost:8000/se :

```
@if(count($lista) == 0)
      <p>Não existe dados na lista</p>
@else
      <p>Existem dados na lista</p>
@endif
```

RECURSOS DE LAYOUT

- Acesse o arquivo de rotas, atualize a rota se e acesse http://localhost:8000/se :

```
Route::get('/se', function() {
    return view('vazio', [
        'lista' => ['item']
    ]);
});
```

RECURSOS DE LAYOUT

- Exemplo: Acesse o arquivo de rotas e registre a seguinte rota:

```
Route::get('/para-cada', function() {
    return view('dados', [
        'lista' => ['dado1', 'dado2']
    ]);
});
```

RECURSOS DE LAYOUT

- Agora edite a view dados.blade.php e acesse http://localhost:8000/para-cada :

```
@foreach($lista as $item)
    <p>{{ $item }}</p>
@endforeach
```

RECURSOS DE LAYOUT

- Para permitir o reuso de código, geralmente dividimos nossas Views em pequenas partes e, depois, as utilizamos quando necessário;

- Assim, é possível alterar partes de View em um só lugar e refletir a alteração em vários lugares ao mesmo tempo. O Blade nos ajuda nesse sentido;

RECURSOS DE LAYOUT

- Exemplo: Acesse o arquivo de rotas e registre a seguinte rota:

```
Route::get('/include', function() {
    return view(‘include‘);
});
```

RECURSOS DE LAYOUT

- Crie na pasta /resources/views a view include.blade.php e acesse http://localhost:8000/include :

```
<h1>Parte da View principal</h1>
@include('welcome')
```

RECURSOS DE LAYOUT

- O Blade também fornece estruturas de repetição. Ex:

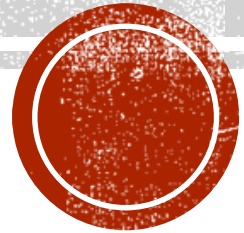
```
@for ($i=0;$i<10;$i++)
    <p>O valor de i é {{ $i }}</p>
@endfor
```

RECURSOS DE LAYOUT

- O Blade também fornece estruturas de repetição. Ex:

```
@while(true)
    <p>Vou travar seu navegador!</p>
@endwhile
```

VALIDAÇÕES DE DADOS



VALIDAÇÕES DE DADOS

- O Laravel nos propõe facilidades para a validação de dados a partir de controllers;

- Para apresentar o referido recurso, vamos implementar, na forma tradicional, um controller com métodos de validação. Logo em seguida, iremos refatorar a classe com recursos de validação do Laravel;

VALIDAÇÕES DE DADOS

- Exemplo: Vá no seu terminal, acesse a partir dele a pasta C:\xampp\htdocs\laravel e digite o comando:

```
php artisan make:controller ValidacaoController
```

- Um novo controller com o nome ValidacaoController será criado em app/Http/Controllers;

VALIDAÇÕES DE DADOS

- Continuando o exemplo, no controller ValidacaoController, crie o seguinte método:

```
public function validaTitulo(Request $request)     {
      if($request->input('titulo') == '')
            return 'Título não pode ser vazio';
      if(strlen($request->input('titulo')) < 3)
            return 'Título deve ter no mínimo 3 letras';
}
```

VALIDAÇÕES DE DADOS

- Agora, no controller ValidacaoController, iremos refatorar o primeiro if do método validaTitulo:

```
public function validaTitulo(Request $request)     {
      $this->validate($request, [
            'titulo' => 'required'
      ]);
      if(strlen($request->input('titulo')) < 3)
            return 'Título deve ter no mínimo 3 letras';
}
```

VALIDAÇÕES DE DADOS

- Em seguida, no controller ValidacaoController, iremos refatorar a classe, tirando a necessidade do segundo if do método validaTitulo:

```
public function validaTitulo(Request $request)     {
      $this->validate($request, [
            'titulo' => 'required|min:3'
      ]);
}
```

VALIDAÇÕES DE DADOS

- Para testar nossas validações, acesse o arquivo de rotas e registre a seguinte rota:

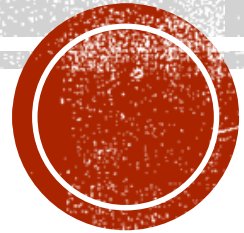
```
Route::get('/form', function() {
    return view(‘form‘);
});
Route::post(‘/form’,
‘ValidacaoController@validaTitulo’);
```

VALIDAÇÕES DE DADOS

- Agora, começaremos a manipular a camada de visualização de dados. Crie a view form.blade.php na pasta resources/views e acesse http://localhost:8000/form :

```
@if(count($errors) > 0)
        @foreach($errors->all() as $error)
                <p>{{ $error }}</p>
        @endforeach
@endif

<form method="post">
        {{ csrf_field() }}
        <p>Titulo: <input type="text" name="titulo"></p>
        <input type="submit" value="Enviar">
</form>
```

CONFIGURANDO BD NO LARAVEL



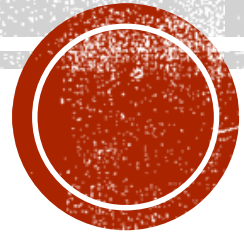
CONFIGURANDO BD

- Crie a base de dados "laravel" a partir do phpmyadmin;

- Abra o arquivo .env localizado na pasta raiz do projeto Laravel e edite as configurações padrão de conexão;

```
5   DB_CONNECTION=mysql
6   DB_HOST=127.0.0.1
7   DB_DATABASE=laravel
8   DB_USERNAME=root
9   DB_PASSWORD=
```

MIGRAÇÕES



MIGRAÇÕES

- Funcionalidade nativa do Laravel;
  - O primeiro framework PHP a se preocupar com isso;
  - Algo que era disponível apenas no Ruby!

- Permite a criação de classes que permitem inicializar (subir) bancos de dados de aplicações e remover informações de bancos quando necessário;

- Facilita a sincronização de dados em banco;

MIGRAÇÕES

- Exemplo: Vá no seu terminal, acesse a partir dele a pasta C:\xampp\htdocs\nomeProjeto e digite o comando:

```
php artisan make:migration tabela_livro --create=livro
```

- Uma nova migration com o nome xxxx_tabela_livro será criada em database/migrations/;

MIGRAÇÕES

- Continuando o exemplo, na migration tabela_livro, edite só o método up:

```
public function up() {
      Schema::create('livro', function(Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('nome');
            $table->timestamps();
      });
}
```

MIGRAÇÕES

- Exemplo: Vá no seu terminal, acesse a partir dele a pasta C:\xampp\htdocs\nomeProjeto e digite o comando:

```
php artisan migrate
```

- Quando o comando for concluído, abra o phpmyadmin, banco laravel e verifique se a tabela livros foi criada com sucesso;

MIGRAÇÕES

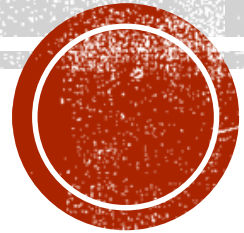
- Exemplo: Para executar o método down da migration, vá no seu terminal, acesse a partir dele a pasta C:\xampp\htdocs\nomeProjeto e digite o comando:

```
php artisan migrate:reset
```

- Quando o comando for concluído, abra o phpmyadmin, banco laravel e verifique se a tabela livros foi excluída com sucesso;

ELOQUENT ORM

Chegou a hora de criar nossas classes Model!!



ELOQUENT ORM

- O Laravel trabalha com ORM (Object Relacional Mapping), que se encarrega de abstrair a conversão das estruturas de tabelas de BD e instruções SQL para o mundo da orientação a objetos;

- O Eloquent implementa o padrão Active Record, em que cada tabela deve possuir uma classe correspondente (um Model MVC);

ELOQUENT ORM

- Exemplo: Vá para a pasta app do projeto laravel e crie a pasta Models (infelizmente o Laravel não cria essa pasta por padrão);
- Vá no seu terminal, acesse a partir dele a pasta C:\xampp\htdocs\nomeProjeto e digite o comando:

```
php artisan make:model Models/LivroModel
```

- Uma nova model com o nome LivroModel será criada em app/Models/;

ELOQUENT ORM

- Continuando o exemplo, edite a Model LivroModel:

```
class LivroModel extends Model {
    protected $table = 'livro';
    protected $primaryKey = 'id';
    protected $fillable = ['nome'];
}
```

ELOQUENT ORM

- Para testar nossas consultas, vamos utilizar uma rota e um controller já definidos anteriormente:

```
Route::resource('livro', 'LivroController');
```

ELOQUENT ORM

- Continuando o exemplo, inclua uma importação na classe LivroController:

```
use App\Models\LivroModel;
```

ELOQUENT ORM

- Em seguida, edite o método index da classe LivroController:

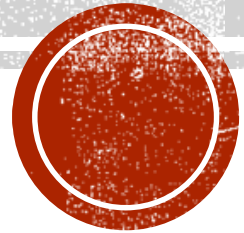
```
public function index() {
    return LivroModel::all();
    /* all(): Método do Eloquent para o
select * from livro  */
}
```

ELOQUENT ORM

- Por fim, abra o phpmyadmin, acesse a tabela livros, insira alguns livros (clicando em Inserir), abra o navegador e acesse http://localhost:8000/livro/

- Um select * from livros será executado automaticamente e apresentado na tela!

ATIVIDADE PRÁTICA



ATIVIDADE PRÁTICA

- Edite o exemplo anterior e faça com que o projeto Laravel também permita inserir, editar e excluir livros, além de exibir informações de um livro específico;

- A sintaxe para os métodos de Model para as devidas ações CRUD estão disponíveis em https://laravel.com/docs/5.*/eloquent

- Faça os registros de rota e a criação de Views necessárias para cada operação;

ATIVIDADE PRÁTICA

- Observação:
  - Para criar um método que possa capturar os inputs do formulário, utilize um objeto da classe Request como parâmetro do método. Exemplo:

```
public function postNome(Request $request) {
        printf("Nome: %s", ($request->input('nome')));
}
```

ATIVIDADE PRÁTICA

- Lembre-se também que cada método CRUD criado em um controller de recurso pode ser executado automaticamente, dependendo da requisição HTTP realizada e do formato da URI:

Tipo	URI	Ação
GET	/livro	index
GET	/livro/create	create
POST	/livro	store
GET	/livro/{id}	show
GET	/livro/{id}/edit	edit
PUT/PATCH	/livro/{id}	update
DELETE	/livro/{id}	destroy

ATIVIDADE PRÁTICA

- Lembre-se também que cada método CRUD criado em um controller de recurso tem uma funcionalidade específica sugerida:

- index – Método que chamará uma view para a exibição de dados;
- create – Método que chamará uma view para o formulário de inserção de dados;
- store – Método que executará a inserção de dados a partir de um Model;
- show - Método que chamará uma view para a exibição de um recurso específico;
- edit - Método que chamará uma view para o formulário de edição de dados;
- update - Método que executará a edição de dados a partir de um Model;
- destroy – Método que executará a exclusão de dados a partir de um Model;

ELOQUENT ORM

- Exemplo: Edite o método store da classe LivroController:

```
public function store(Request $request) {
      $livro = new LivroModel;
      $livro->nome = $request->input('nome');
      $livro->save();

      return LivroModel::all();
}
```

ELOQUENT ORM

- Exemplo: Edite o método create da classe LivroController:

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

ELOQUENT ORM

- Exemplo: Crie a view cadastro.blade.php para o cadastro de livros:

```
<html>
<body>
      <form method="post" action="/livro">
            {{ csrf_field() }}
            <p>Nome: <input type="text" name="nome"></p>
      </form>
</body>
</html>
```

ELOQUENT ORM

- Abra o navegador e acesse http://localhost:8000/livro e http://localhost:8000/livro/create para testar as rotas baseadas em métodos do nosso controller de recurso;

ELOQUENT ORM

- Exemplo: Edite o método update da classe LivroController:

```
public function update(Request $request, $id) {
      $livro = LivroModel::find($id);
      $livro->nome = $request->input('nome');
      $livro->save();

      return LivroModel::all();
}
```

ELOQUENT ORM

- Exemplo: Edite o método edit da classe LivroController:

```
public function edit($id) {
     return view('edita', [
          'id' => $id
     ]);
}
```

ELOQUENT ORM

- Exemplo: Crie a view edita.blade.php para a edição de livros:

```
<html>
<body>
        <form method="post" action="/livro/{{ $id }}">
                {{ csrf_field() }}
                {{ method_field('PUT') }}
                <p>Nome: <input type="text" name="nome"></p>
        </form>
</body>
</html>
```

ELOQUENT ORM

- Abra o navegador e acesse http://localhost:8000/livro/1/edit para testar as rotas baseadas em métodos do nosso controller de recurso;

ELOQUENT ORM

- Exemplo: Edite o método edit da classe LivroController:

```
public function edit($id) {
     return view('edita', [
          'id' => $id,
          'nome' => $this->show($id)->nome
     ]);
}
```

ELOQUENT ORM

- Exemplo: Crie a view edita.blade.php para a edição de livros:

```
<html>
<body>
	<form method="post" action="/livro/{{ $id }}">
		{{ csrf_field() }}
		{{ method_field('PUT') }}
		<p>Nome: <input type="text" name="nome"
value="{{$nome}}"></p>
	</form>
</body>
</html>
```

ELOQUENT ORM

- Exemplo: Edite o método show da classe LivroController:

```
public function show($id) {
    return LivroModel::find($id);
}
```

ELOQUENT ORM

- Abra o navegador e acesse http://localhost:8000/livro/1/edit para testar as rotas baseadas em métodos do nosso controller de recurso;

ELOQUENT ORM

- Exemplo: Edite o método destroy da classe LivroController:

```
public function destroy($id) {
      $livro = LivroModel::find($id);
      $livro->delete();

      return LivroModel::all();
}
```

ELOQUENT ORM

- Exemplo: Atuaize a view edita.blade.php para a exclusão de livros:

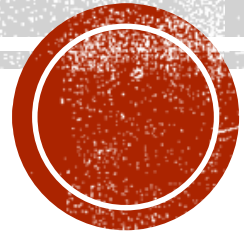
```
<html>
<body>
        ...
        <form method="post" action="/livro/{{ $id }}">
                {{ csrf_field() }}
                {{ method_field('DELETE') }}
                <p><input type="submit" value="EXCLUIR"></p>
        </form>

</body>
</html>
```

ELOQUENT ORM

- Abra o navegador e acesse http://localhost:8000/livro/1/edit e clique em excluir para testar as rotas baseadas em métodos do nosso controller de recurso;

AUTENTICAÇÃO

Que tal criar telas de registro e login facilmente?



AUTENTICAÇÃO

- O Laravel torna a implementação de autenticação muito simples (a partir da versão 5.3);

- Na verdade, quase tudo já está configurado no framework para tal, bastando que você adapte os formulários de registro e de login gerados automaticamente pelo Laravel;

AUTENTICAÇÃO

- Como exemplo, vamos criar um novo projeto Laravel a partir do comando:

```
composer create-project laravel/laravel projetoLogin "5.8"
```

- Uma vez instalado, edite o arquivo .env contido na pasta raiz do projeto:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=sistema
DB_USERNAME=root
DB_PASSWORD=
```

AUTENTICAÇÃO

- Em projetos Laravel, existem migrations já relacionadas com tabelas SQL para autenticação de usuários. Para criá-las no nosso banco, execute o comando:

```
php artisan migrate
```

- Para criar toda a estrutura de login e senha, execute o comando:

```
php artisan make:auth
```

AUTENTICAÇÃO

- Ative o servidor:

```
php artisan serve
```

- Acesse o link http://localhost:8000/register e crie o primeiro usuário da sua aplicação!

AUTENTICAÇÃO

- Para proteger rotas e recursos da aplicação a partir da autenticação realizada, acesse o arquivo de rotas (routes/web.php) e adicione o middleware auth em cada rota. Exemplo:

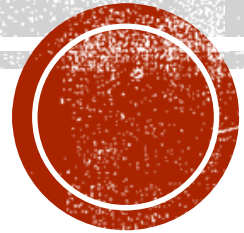
```
Route::get('/cadastro', function() {
    echo 'Precisa de Login';
})->middleware('auth');
```

AUTENTICAÇÃO

- Para proteger rotas e recursos da aplicação implementados em um Controller, crie um construtor para o mesmo e adicione uma chamada de middleware auth. Exemplo:

```
public function __construct() {
    $this->middleware('auth');
}
```

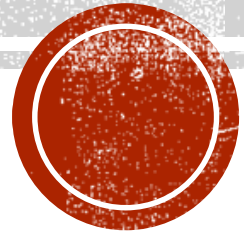
ATIVIDADE PRÁTICA



ATIVIDADE PRÁTICA

- Atualize a aplicação da calculadora criada anteriormente, fazendo com que o usuário precise estar logado para acessar seus recursos;
  - OBS. 1: Aproveite a estrutura da View da tela de Dashboard (resources/views/home.blade) e faça com que todas as telas da aplicação sigam o mesmo padrão!
  - OBS. 2: Fique a vontade para definir as rotas no arquivo de rotas ou a partir de um Controller, contanto que não seja possível o acesso a rotas sem autenticação!

ATIVIDADE PRÁTICA



ATIVIDADE PRÁTICA

- Atualize a aplicação para o cadastro de livros criada parcialmente na atividade anterior, fazendo com que o usuário precise estar logado para acessar seus recursos;
  - OBS. 1: Aproveite a estrutura da View da tela de Dashboard (resources/views/home.blade) e faça com que todas as telas da aplicação sigam o mesmo padrão!
  - OBS. 2: Fique a vontade para definir as rotas no arquivo de rotas ou a partir de um Controller, contanto que não seja possível o acesso a rotas sem autenticação!
  - OBS. 3: Crie um menu na página de Dashboard que permita ao usuário acessar todas as Views e/ou operações de CRUD criadas no Controller LivroController;

LOCALIZAÇÃO

- O Laravel provê uma forma conveniente para a geração de strings em várias linguagens, permitindo o suporte a múltiplas linguagens em suas aplicações;
- Strings de linguagem são armazenados e definidos no diretório resources/lang;
- Neste diretório é possível criar um subdiretório para cada linguagem suportada por uma aplicação Laravel;
- A linguagem padrão da sua aplicação Laravel é definida no arquivo config/app.php, índice locale;

```
/resources
   /lang
      /en
         messages.php
      /es
         messages.php
```

LOCALIZAÇÃO

- Exemplo: baixe o diretório de linguagem pt-BR em: **https://tinyurl.com/yynvqreb** ;

- Descompacte a pasta pt-BR na pasta resources/lang;

- Abra o arquivo config/app.php e substitua o conteúdo do elemento 'locale' para 'pt-BR';

LOCALIZAÇÃO

- Exemplo: Vá no seu terminal, acesse a partir dele a pasta C:\xampp\htdocs\laravel e digite o comando:

```
php artisan make:controller ValidacaoController
```

- Um novo controller com o nome ValidacaoController será criado em app/Http/Controllers;

LOCALIZAÇÃO

- Em seguida, no controller ValidacaoController, implemente o método validaTitulo:

```
public function validaTitulo(Request $request)
{
      $this->validate($request, [
            'titulo' => 'required|min:3'
      ]);
}
```

LOCALIZAÇÃO

- Para testar nossas validações, acesse o arquivo de rotas e registre a seguinte rota:

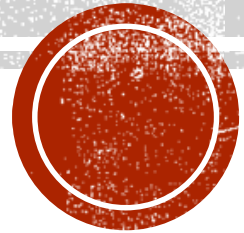
```
Route::get('/form', function() {
    return view(‘form‘);
});
Route::post(‘/form’,
‘ValidacaoController@validaTitulo’);
```

LOCALIZAÇÃO

- Agora, começaremos a manipular a camada de visualização de dados. Crie a view form.blade.php na pasta resources/views e acesse http://localhost:8000/form :

```
@if(count($errors) > 0)
        @foreach($errors->all() as $error)
                <p>{{ $error }}</p>
        @endforeach
@endif

<form method="post">
        {{ csrf_field() }}
        <p>Titulo: <input type="text" name="titulo"></p>
        <input type="submit" value="Enviar">
</form>
```

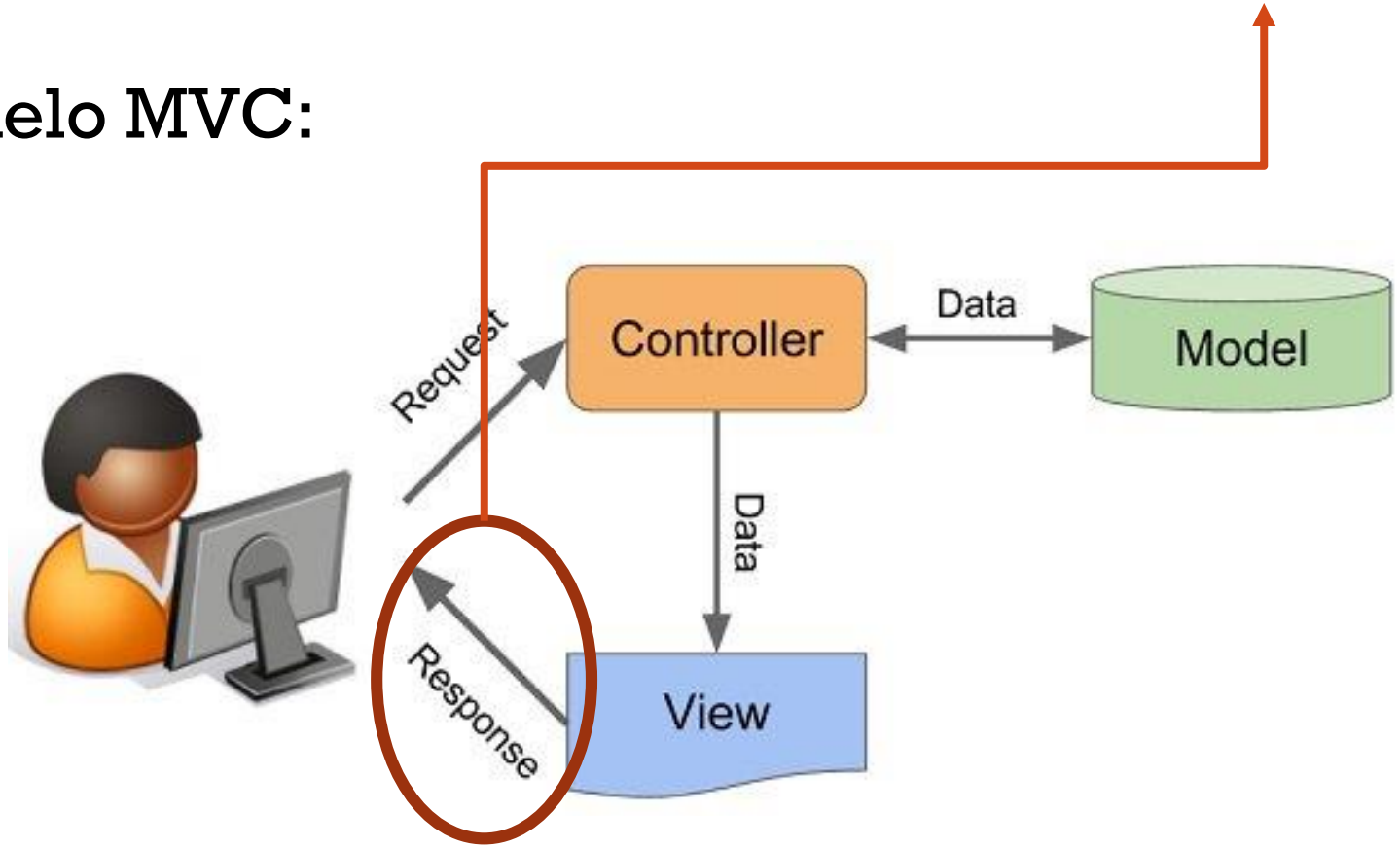
RESPOSTAS HTTP



LARAVEL

O Lavarel também permite gerar e manipular respostas HTTP.

- O modelo MVC:



RESPOSTAS HTTP

- A forma mais básica para a geração de respostas HTTP no Laravel é a partir do comando return:

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

- Porém, retornos no Laravel podem ser bem mais inteligentes, graças a recursos fonecidos pelo framework;

RESPOSTAS HTTP

- Respostas do tipo JSON
  - Por padrão, o Laravel já retorna arrays no formato JSON:

```
Route::get('/ola',  function() {
    return ['Thyago', 25];
});
```

RESPOSTAS HTTP

- Respostas do tipo JSON
  - Porém, é possível explicitar uma resposta JSON a partir do método response():

```
Route::get('/ola',  function() {
    return response()->json(['Thyago', 25]);
});
```

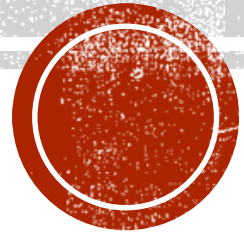
RESPOSTAS HTTP

- Inserindo cabeçalhos em Respostas HTTP
  - Exemplo: Como criar um arquivo CSV a partir de dados gerados no Laravel?

```
Route::get('/downloadCSV', function() {
    $times = ['Sport', 'Nautico', 'Santa Cruz'];
    $content = implode("\n", $times);
    return response($content)
    ->header('Content-type','application/force-download')
    ->header('Content-
Disposition','attachment;filename="times.csv"');
});
```

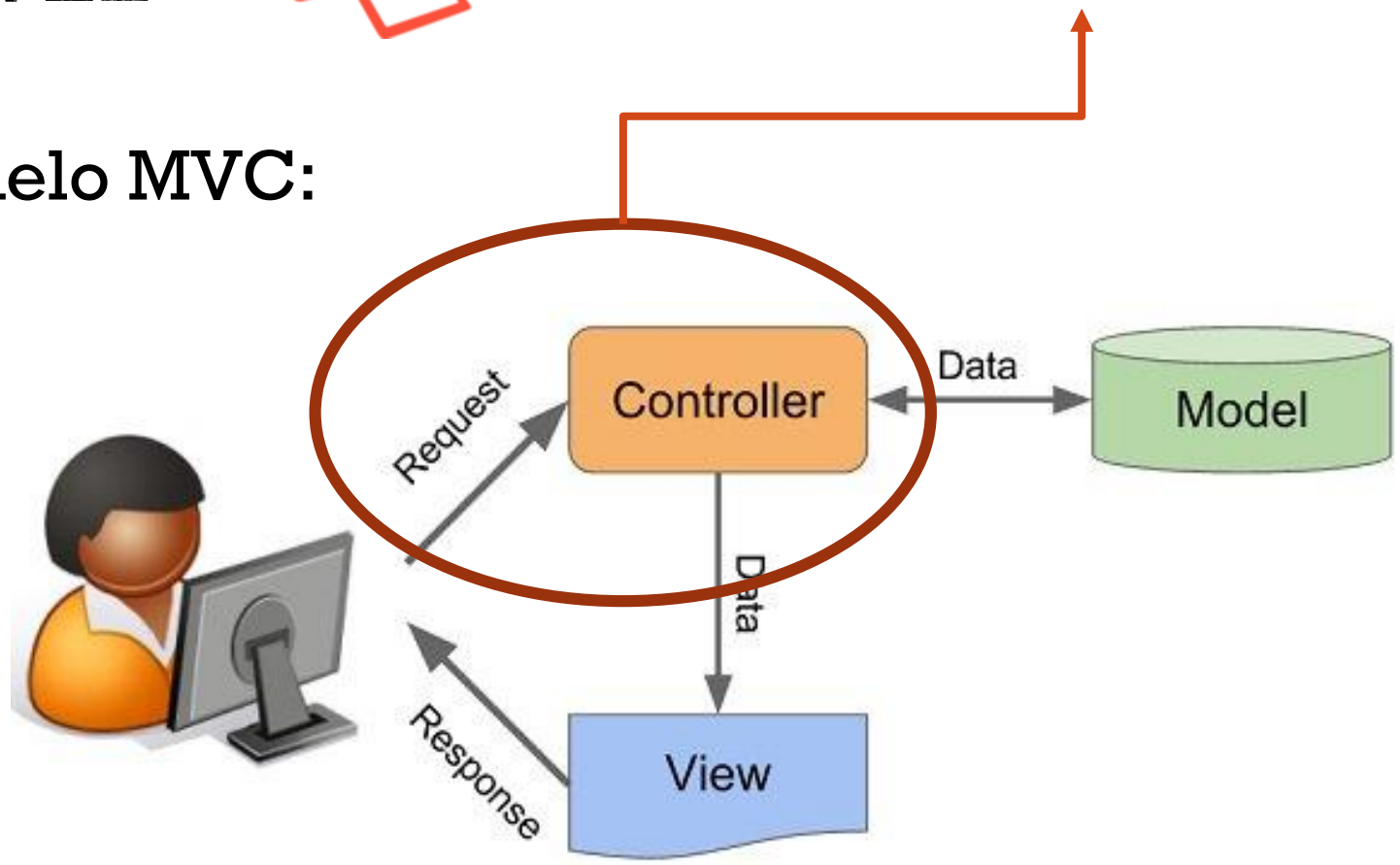
FORM REQUEST

Uma forma de validar dados mais eficiente



LARAVEL

- O modelo MVC:



FORM REQUEST

- Tratam- se de controladores com o objetivo de validar dados de requisições HTTP;
- Tornam os códigos de validação mais limpos e fáceis de manter;
- Com eles, podemos invocar módulos da nossa aplicação apenas se os dados a serem submetidos aos mesmos forem válidos;

FORM REQUEST

- Podemos utilizar o artisan para criar uma classe PHP que modela um Form Request na pasta app/Http/Requests:

```
php artisan make:request LivroFormRequest
```

- Na classe criada, será definido automaticamente o método rules(), onde deveremos implementar todas as regras de validação;

- Além disso, será criado o método authorize(), que retorna um valor booleano para a autorizar ou não usuários a fazerem requisições a partir do Form Request;

FORM REQUEST

- Exemplo:

```
public function authorize() {
        return true;
}
public function rules() {
        return [
                'titulo'     => 'required',
                'autor'      => 'required|min:3'
        ];
}
```

FORM REQUEST

- Exemplo: Vamos criar na pasta /resources/views a view formlivro.blade.php:

```
@extends('layouts.app')
@section('content')
        <div class="container">
                <form method="post">
                {{ csrf_field() }}
                Titulo: <input type="text" name="titulo">
                Autor: <input type="text" name="autor">
                <input type="submit" value="Submeter">
                </form>
        </div>
@endsection
```

Digite no cmd o comando php artisan make:auth para a criação Da view app.blade.php

FORM REQUEST

- Crie o controller LivroController na pasta app/Http/Controllers:

```
php artisan make:controller LivroController
```

- Na classe criada, importe o Form Request LivroFormRequest:

```
use App\Http\Requests\LivroFormRequest;
```

FORM REQUEST

- Ainda no controller LivroController, insira o seguinte método:

```
public function checar(LivroFormRequest
$request) {
    return $request->all();
}
```

FORM REQUEST

- Continuando nosso exemplo, acesse o arquivo de rotas, adicione as seguintes rotas e acesse http://localhost:8000/formlivro :

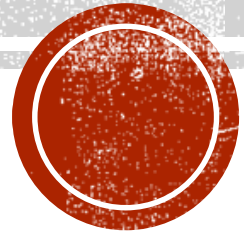
```
Route::get('/formlivro', function() {
      return view('formlivro');
});
Route::post('/formlivro', 'LivroController@checar');
```

FORM REQUEST

- Para apresentar os erros de validação existentes, podemos incluir o seguinte bloco Blade na view formlivro.blade.php:

```
@if(count($errors) > 0)
        @foreach($errors->all() as $error)
                <p>{{ $error }}</p>
        @endforeach
@endif
```

ENVIANDO E-MAILS



ENVIANDO E-MAILS

- O Laravel torna o envio de e-mails uma tarefa trivial;
- A partir de algumas configurações e aplicações de serviços, é possível o envio de e-mails via Laravel sem nenhuma biblioteca externa;
- Como exemplo, utilizaremos o mailtrap.io, que fornece uma maneira simples de criar uma caixa para o envio e recebimento de e-mails;

ENVIANDO E-MAILS

- Crie uma conta gratuita no mailtrap.io;
- Clique na caixa Demo inbox;
- Insira as credenciais apresentadas no mailtrap no arquivo .env localizado na raiz da sua aplicação Laravel;

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

ENVIANDO E-MAILS

- Crie o controller EmailSimplesController na pasta app/Http/Controllers:

```
php artisan make:controller EmailSimplesController
```

- Na classe criada, importe a classe Mail (use Mail;) e crie o seguinte método:

```
public function email() {
      return view('email_simples');
}
```

ENVIANDO E-MAILS

- Crie a view email_simples.blade.php:

```
<div class="container">
      <form method="post">
       {{ csrf_field() }}
      Assunto: <input type="text" name="assunto">
      Mensagem: <textarea name="mensagem"></textarea>
      <input type="submit" value="Submeter">
      </form>
</div>
```

ENVIANDO E-MAILS

- Continuando nosso exemplo, acesse o arquivo de rotas, adicione a seguinte rota e acesse http://localhost:8000/email/simples :

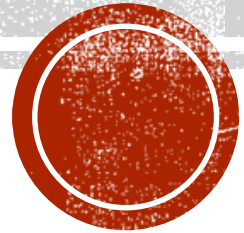
```
Route::match(['GET','POST'], '/email/simples',
'EmailSimplesController@email');
```

ENVIANDO E-MAILS

- Atualize o método email() do controller EmailSimplesController:

```
public function email(Request $request) {
        if($request->isMethod('post')) {
          $assunto = $request->input('assunto');
          $mensagem = $request->input('mensagem');
          Mail::raw($mensagem, function($swiftMessage) use ($assunto) {
            $swiftMessage->subject($assunto);
            $swiftMessage->to('thyagomaia@gmail.com');
          });
        }
        return view('email_simples');
}
```

SERVICE PROVIDERS



SERVICE PROVIDERS

- Já utilizamos um Service Provider anteriormente, quando tivemos a necessidade de refatorar a classe Schema do Laravel, alterando a forma como a mesma cria campos Varchar em bancos MySQL a partir da classe app/Providers/AppServiceProvider;

- Um Provider é uma classe Laravel que permite a integração de APIs ou Classes externas, sem a necessidade de realizar grandes adaptações em interfaces e/ou procedimentos nos módulos exportados;

SERVICE PROVIDERS

- Toda adaptação ou operações de integração necessárias serão implementadas no Provider;

- O Provider ficará responsável pela interação com APIs e Classes externas, abstraindo sua integração em um componente de software dedicado para tal;

SERVICE PROVIDERS

- Exemplo: Você já tem um sistema Laravel com todas as operações de CRUD implementadas para um banco de dados específico;

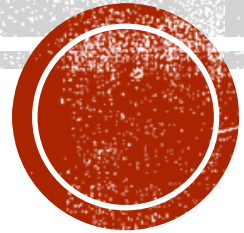
- Porém houve a necessidade de mudar de banco, com uma API de conexão e consulta completamente diferente. Isso significa refatorar todas as ocorrências de cada operação para a nova API. Um Provider facilita e centraliza esse processo;

SERVICE PROVIDERS

- Listas de Providers são definidos no arquivo config/app.php;

```
'providers' => [

    /*
     * Laravel Framework Service Providers...
     */
    Illuminate\Auth\AuthServiceProvider::class,
    Illuminate\Broadcasting\BroadcastServiceProvider::class,
    Illuminate\Bus\BusServiceProvider::class,
    Illuminate\Cache\CacheServiceProvider::class,
    Illuminate\Foundation\Providers\ConsoleSupportServiceProvider::class,
    Illuminate\Cookie\CookieServiceProvider::class,
    Illuminate\Database\DatabaseServiceProvider::class,
    Illuminate\Encryption\EncryptionServiceProvider::class,
    Illuminate\Filesystem\FilesystemServiceProvider::class,
    Illuminate\Foundation\Providers\FoundationServiceProvider::class,
    Illuminate\Hashing\HashServiceProvider::class,
    Illuminate\Mail\MailServiceProvider::class,
    Illuminate\Notifications\NotificationServiceProvider::class,
    Illuminate\Pagination\PaginationServiceProvider::class,
    Illuminate\Pipeline\PipelineServiceProvider::class,
    Illuminate\Queue\QueueServiceProvider::class,
    Illuminate\Redis\RedisServiceProvider::class,
    Illuminate\Auth\Passwords\PasswordResetServiceProvider::class,
    Illuminate\Session\SessionServiceProvider::class,
    Illuminate\Translation\TranslationServiceProvider::class,
    Illuminate\Validation\ValidationServiceProvider::class,
    Illuminate\View\ViewServiceProvider::class,
```

MINIPROJETO – GERADOR DE QR CODE



MINIPROJETO

- Como exemplo, vamos criar um novo projeto Laravel a partir do comando:

```
composer create-project laravel/laravel
LaravelQR "5.8.*"
```

- Logo em seguida, iremos instalar o pacote simple-qrcode no nosso projeto:

```
composer require simplesoftwareio/simple-qrcode
```

MINIPROJETO

- Após a instalação do pacote Simple QR Code, abra o arquivo config/app.php e adicione o provedor e aliase do serviço:

```
//config/app.php

'providers' => [
….
SimpleSoftwareIO\QrCode\QrCodeServiceProvider::class
],

'aliases' => [
….
'QrCode' => SimpleSoftwareIO\QrCode\Facades\QrCode::class
],
```

MINIPROJETO

- Logo em seguida, no arquivo routes/web.php, vamos configurar a seguinte rota:

```
Route::get('qrcode', function () {
     return QrCode::size(300)->generate('A basic example of QR code!');
 });
```

MINIPROJETO

- Acesse http://localhost:8000/qrcode e verifique o QR Code gerado;
- Agora, no arquivo routes/web.php, vamos configurar a seguinte rota para a geração de um QR Code colorido:

```
Route::get('qrcode-with-color', function () {
     return \QrCode::size(300)
                        ->backgroundColor(255,55,0)
                        ->generate('A simple example of QR code');
 });
```

MINIPROJETO

- Acesse http://localhost:8000/qrcode-with-color e verifique o QR Code gerado;
- Agora, no arquivo routes/web.php, vamos configurar a seguinte rota para a geração de um QR Code com imagem (salve uma imagem na pasta public da sua aplicação):

```
Route::get('qrcode-with-image', function () {
        $image = \QrCode::format('png')
                        ->merge('images/laravel.png', 0.5, true)
                        ->size(500)->errorCorrection('H')
                        ->generate('A simple example of QR code!');
    return response($image)->header('Content-type','image/png');
});
```

MINIPROJETO

- Acesse http://localhost:8000/qrcode-with-image e verifique o QR Code gerado:

MINIPROJETO

- Agora, no arquivo routes/web.php, vamos configurar a seguinte rota para a geração de um E-Mail QR Code:

```
Route::get('qrcode-with-special-data', function() {
     return \QrCode::size(500)
                  ->email('info@tutsmake.com', 'Welcome to Tutsmake!', 'This is !.');
 });
```

MINIPROJETO

- Acesse http://localhost:8000/qrcode-with-special-data e verifique o QR Code gerado;
- Agora, no arquivo routes/web.php, vamos configurar uma rota para a geração de um QR Code Phone Number a partir do seguinte método:

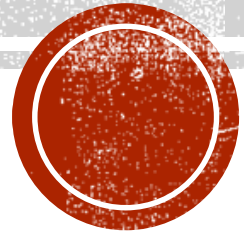
```
QrCode::phoneNumber('111-222-6666');
```

MINIPROJETO

- Também é possível a criação de QR Codes para o envio automático de SMSs:

```
QrCode::SMS('111-222-6666', 'Body of the message');
```

LOGGING NO LARAVEL



LOGGING

- Assim como em outros recursos do Laravel, é possível gerar arquivos de log de uma maneira bastante simples;
- O Laravel também fornece a biblioteca Monolog, que provê ferramentas poderosas para a manipulação de logs;
- Vamos aproveitar nosso sistema de Help Desk criado anteriormente para exemplificar o registro de logs no Laravel;

LOGGING

- Abra o controller TicketsController e adicione o código abaixo:

```
use Illuminate\Support\Facades\Log;

public function index() {
	$tickets = Ticket::paginate(10);
	$categories = Category::all();
	Log::info('Um usuário acessou todos os chamados gerados');

	return
view('tickets.index',compact('tickets','categories'));
}
```

LOGGING

- Acesse a aplicação, logando com uma conta administrativa (id_admin = 1) e visualize todos os chamados gerados até então;

- Acesse o arquivo storage/logs/laravel.log associado com o dia dos testes e verifique a última linha do arquivo:

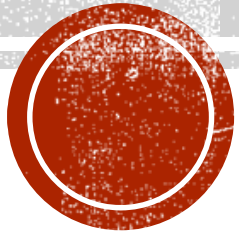
```
[2019-09-26 14:42:25] local.INFO: Um usuário acessou todos os chamados gerados
```

LOGGING

- Níveis de log disponíveis, na ordem de prioridades, a partir da classe Facade Log:

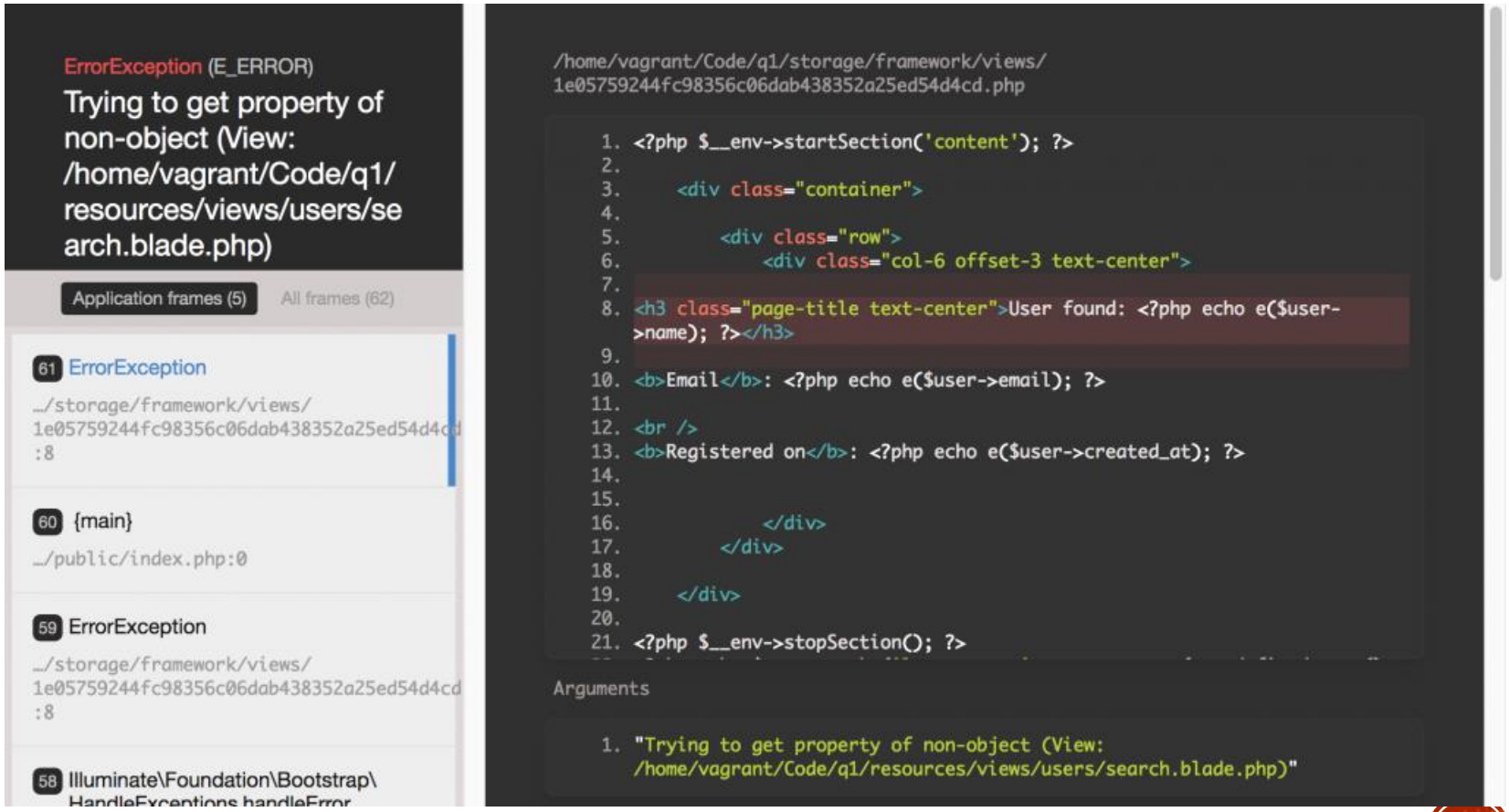
```
Log::emergency($message);
Log::alert($message);
Log::critical($message);
Log::error($message);
Log::warning($message);
Log::notice($message);
Log::info($message);
Log::debug($message);
```

EXCEÇÕES NO LARAVEL



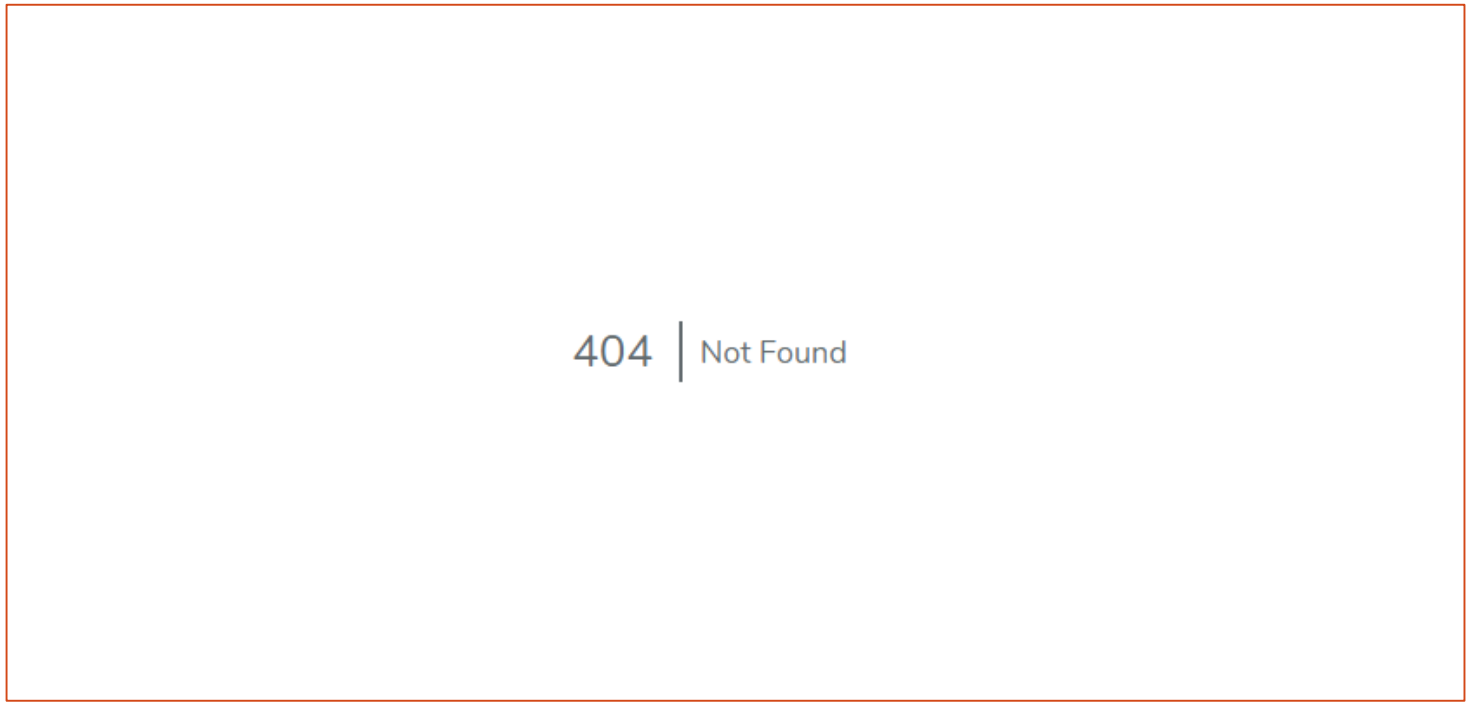
EXCEÇÕES

- Quando algo de errado acontece no Laravel, o framework apresenta uma view similar com a apresentada ao lado:



EXCEÇÕES

- Quando algo de errado acontece no Laravel, o framework apresenta uma view similar com a apresentada ao lado:



EXCEÇÕES

- A view em questão é bastante útil para desenvolvedores, mas nada atrativa para os usuários!

- O Laravel fornece recursos para a manipulação e lançamento de exceções de uma forma mais elegante;

- Vamos aproveitar nosso sistema de Help Desk criado anteriormente para exemplificar a manipulação de exceções no Laravel;

EXCEÇÕES

- Crie a view testeerro.blade.php e adicione o código a seguir:

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

@section('content')
    <div class="row justify-content-center">
        <div class="col-md-8">
            <div class="card">
                <div class="card-header">Buscar chamado por ID</div>

                <div class="card-body">
                    <form action="/busca" method="post">
                    @csrf
                    <input type="text" name="id" class="form-control">
                    </form>
                </div>
            </div>
        </div>
    </div>
@endsection
```

EXCEÇÕES

- Atualize o arquivo web.php, incluindo as seguintes rotas:

```
Route::get('testerro', 'BuscaController@testeErro');
Route::post('busca', 'BuscaController@busca');
```

EXCEÇÕES

- Crie o controller BuscaController e adicione os métodos a seguir:

EXCEÇÕES

```php
use App\Ticket;

class BuscaController extends Controller
{
    public function testeErro() {
        return view('testeerro');
    }

    public function busca(Request $request) {
        $ticket = Ticket::findOrFail($request->input('id'));
        return view('busca', compact('ticket'));
    }
}
```

EXCEÇÕES

- Crie a view busca.blade.php e adicione o código a seguir:

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

@section('content')
    <div class="row justify-content-center">
        <div class="col-md-8">
            <div class="card">
                <div class="card-header">Buscar chamado por ID</div>

                <div class="card-body">
                    <h3>Ticket encontrado: {{ $ticket->ticket_id }}</h3>
                    <p><b>Título:</b> {{ $ticket->title }}</p>
                    <p><b>Mensagem:</b> {{ $ticket->message }}</p>
                </div>
            </div>
        </div>
    </div>
@endsection
```

EXCEÇÕES

- Precisamos atualiza o método de busca do controller BuscaController para que o mesmo esteja preparado para lançar exceções quando um modelo não for encontrado;

- Logo, também precisaremos saber qual classe Laravel é responsável pela manipulação desse tipo de erro. No caso, é a classe ModelNotFoundException;

- Podemos então associar uma instância dessa classe de erro em um bloco try-catch para criar nossos próprios reports;

```php
use App\Ticket;
use Illuminate\Database\Eloquent\ModelNotFoundException;

class BuscaController extends Controller
{
    public function testeErro() {
        return view('testeerro');
    }

    public function busca(Request $request) {
        try {
            $ticket = Ticket::findOrFail($request->input('id'));
        }
        catch(ModelNotFoundException $erro) {
            return back()->withError('Ticket não encontrado');
        }

        return view('busca', compact('ticket'));
    }
}
```

EXCEÇÕES

- Também precisaremos atualizar a view testeerro para reportar um erro quando a exceção for capturada:

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

@section('content')
    <div class="row justify-content-center">
        <div class="col-md-8">
            <div class="card">
                <div class="card-header">Buscar chamado por ID</div>

                <div class="card-body">
                    @if(session('error'))
                        <div class="alert alert-danger">
                        {{ session('error') }}
                        </div>
                    @endif
                    <form action="/busca" method="post">
                    @csrf
                    <input type="text" name="id" class="form-control">
                    </form>
                </div>
            </div>
        </div>
    </div>
@endsection
```

EXCEÇÕES

- Acesse http://localhost:8000/testeerro e verifique se o erro 404 será substituído pela mensagem de erro customizada;