

## SUMÁRIO

<b>1. FUNDAMENTOS DO WORKFLOW DO LARAVEL.....</b>	<b>1</b>
<b>2. ORGANIZAÇÃO INICIAL .....</b>	<b>1</b>
<b>3. O ROTEAMENTO .....</b>	<b>1</b>
3.1. SÍNTEXE BÁSICA DO ROTEAMENTO .....	1
3.2. ROTAS DE VIEWS .....	2
3.3. REDIRECIONAMENTO COM REDIRECT NO ARQUIVO DE ROTA .....	2
3.4. ROTAS COM PARÂMETROS OBRIGATÓRIOS E OPCIONAIS .....	2
3.4.1. ROTA COM PARÂMETROS OBRIGATÓRIOS - EXEMPLO .....	2
3.4.2. ROTA COM PARÂMETROS OPCIONAIS – EXEMPLO .....	2
3.5. ROTAS NOMEADAS .....	3
3.6. MIDDLEWARE .....	3
3.6.1. MATRIZ DOS MIDDLEWARES .....	3
3.6.2. COMO UTILIZAR OS MIDDLEWARES PARA PROTEGER AS ROTAS .....	4
3.7. PROTEÇÃO CONTRA ATAQUES CSRF .....	4
<b>4. OS CONTROLLERS .....</b>	<b>5</b>
4.1. CRIAÇÃO DE UM CONTROLLER .....	5
4.2. CRIANDO UM CONTROLLER SINGLE ACTION .....	5
4.3. CAPTURANDO REQUESTS .....	6
4.4. OS RETORNOS DOS CONTROLLERS.....	6
4.4.1. RETORNAR UMA VIEW .....	6
4.4.2. RETORNAR UM OBJETO RESPONSE.....	7
4.4.3. REDIRECIONAMENTO COM REDIRECT.....	7
4.4.4. TIPO DO DADO DA RESPONSE .....	7
4.5. CONTROLLERS DE RECURSOS: PARA CRUDS GENÉRICOS .....	7
4.5.1. A ORGANIZAÇÃO DE UM CONTROLADOR DE RECURSOS .....	8
4.5.2. CONTROLLER RESOURCE COM MODEL ESPECIFICADO .....	8
4.5.3. CONTROLLER RESOURCE PARA FORMULÁRIOS .....	9
4.5.4. ESPECIFICANDO AS ACTIONS DISPONÍVEIS .....	9
<b>5. AS VIEWS .....</b>	<b>9</b>
5.1. O TEMPLATE BLADE .....	9
5.2. ENVIANDO DADOS PARA AS VIEWS .....	9
5.3. AS DIRETIVAS DO TEMPLATE BLADE.....	10

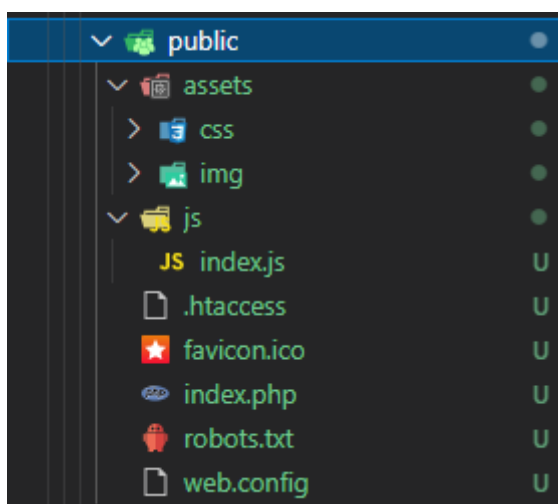
5.3.1.	DIRETIVA @EXTENDS .....	10
5.3.2.	DIRETIVA @SECTION.....	10
5.3.3.	DIRETIVA @YIELD.....	10
5.3.4.	OUTRAS DIRETIVAS .....	10
<b>5.4.</b>	<b>LAYOUTS E HERANÇA: VIEW CONSTANTES E VARIÁVEIS.....</b>	<b>10</b>
<b>6.</b>	<b>SESSÕES .....</b>	<b>11</b>
<b>6.1.</b>	<b>O APLICATIVO LARAVEL JÁ POSSUI UMA SESSÃO ATIVA .....</b>	<b>12</b>
<b>6.2.</b>	<b>CRIANDO UM VALOR DE SESSÃO .....</b>	<b>12</b>
<b>6.3.</b>	<b>RESGATANDO VALOR DA SESSÃO .....</b>	<b>12</b>
6.3.1.	RECUPERANDO DADOS DA SESSION COM A INSTÂNCIA DE REQUEST .....	12
6.3.2.	RECUPERANDO DADOS DA SESSION COM O FACADE SESSION .....	12
<b>7.</b>	<b>SISTEMA DE LOGS .....</b>	<b>13</b>
<b>7.1.</b>	<b>COMO FUNCIONA O SISTEMA DE LOGS .....</b>	<b>13</b>
7.1.1.	ARQUIVO DE CONFIGURAÇÃO DOS LOGS.....	13
7.1.2.	DRIVERS DE CANAL DOS LOGS.....	14
7.1.3.	NÍVEL DE REGISTRO DO CANAL .....	14
7.1.4.	GRAVANDO MENSAGENS COM O FACADE LOG .....	14
7.1.5.	COMO CRIAR UM LOG/CANAL.....	14
<b>FONTE .....</b>		<b>15</b>

## 1. FUNDAMENTOS DO WORKFLOW DO LARAVEL

Esse documento não é diferente da documentação oficial do Laravel, e especificamente da sua seção “basics”, sobre o funcionamento básico no nível do código, exceto pelo fato de que é escrito da forma mais clara possível, e com a adição de informações externas complementares.

## 2. ORGANIZAÇÃO INICIAL

É recomendado, na própria documentação do Laravel, que os arquivos de estilização, o Javascript e imagens sejam também armazenados na pasta public.



## 3. O ROTEAMENTO

Todas as rotas do Laravel são definidas em seus arquivos de rota, que estão localizados no diretório routes. Cada rota é controlada por um controlador, e por uma action/método específica dele.

Existe mais de um arquivo de roteamento, mas os principais podemos considerar como sendo o web.php, que é o roteamento para aplicações web, e o api.php, que serve para o roteamento das requisições AJAX.

### 3.1. SÍNTEXE BÁSICA DO ROTEAMENTO

Para a maioria dos aplicativos, você começará definindo rotas no arquivo routes/web.php. As rotas definidas em routes/web.php podem ser acessadas inserindo a URL da rota definida em seu navegador.

A sintaxe é essa básica para a configuração de uma rota é essa:

```
use App\Http\Controllers\UserController;

Route::get('/user', [UserController::class, 'index']);

Route::get("/user", "UserController@index");
```

No exemplo acima, a rota “/user” é controlada pelo controller “UserController”, e pela sua action “index”, via método GET. Para rotas de processamento de formulários, por exemplo, usa-se o método Route::post.

### 3.2. ROTAS DE VIEWS

Se a rota configurada tem o único papel de retornar uma view, não é preciso criar um método específico em um controlador para fazer isso, mas basta que se crie uma rota de visualização.

```
Route::view('/welcome', 'welcome'); // Sem envio de dados
```

```
Route::view('/welcome', 'welcome', ["tittle" => "Welcome"]); // Com envio de dados
```

### 3.3. REDIRECIONAMENTO COM REDIRECT NO ARQUIVO DE ROTA

Se você está definindo uma rota que redireciona para outro URI, você pode usar o método Route::redirect. Este método fornece um atalho conveniente para que você não precise definir uma rota completa ou controlador para realizar um redirecionamento simples:

```
Route::redirect('/origem', '/destino');
```

### 3.4. ROTAS COM PARÂMETROS OBRIGATÓRIOS E OPCIONAIS

**É possível incluir parâmetros obrigatórios ou opcionais** na configuração das rotas, fazendo com que sejam recuperados por uma call-back, ou como argumento na action do controlador.

#### 3.4.1. ROTA COM PARÂMETROS OBRIGATÓRIOS - EXEMPLO

```
Route::get("/user/{a}/{b}", "UserController@index");

public function index($a, $b){ // }
```

#### 3.4.2. ROTA COM PARÂMETROS OPCIONAIS – EXEMPLO

Se o parâmetro é opcional, a variável que irá recebê-lo, ou não, deve ter um valor default.

```
Route::get("/user/{a}/{b?}", "UserController@index");

public function index($a, $b = null){ // }
```

### 3.5. ROTAS NOMEADAS

As rotas nomeadas permitem a geração conveniente de URLs ou redirecionamentos para rotas específicas. Você pode especificar um nome para uma rota encadeando o `name()` na definição de rota:

```
use App\Http\Controllers\UserController;

Route::get('/user', [UserController::class, 'index'])->name("user");
```

Assim, em métodos que utilizam as rotas do aplicativo, como o `redirect()`, ao invés de informar a rota com a inclusão da barra, `/`, pode ser informado apenas o seu nome – neste caso seria `redirect("user")`.

### 3.6. MIDDLEWARE

O middleware fornece um mecanismo conveniente para inspecionar e filtrar solicitações HTTP que entram em seu aplicativo.

Por exemplo, o Laravel inclui um middleware que verifica se o usuário do seu aplicativo está autenticado. Se o usuário não estiver autenticado, o middleware redirecionará o usuário para a tela de login do seu aplicativo.

Middleware adicional pode ser escrito para executar uma variedade de tarefas além da autenticação.

#### 3.6.1. MATRIZ DOS MIDDLEWARES

Todos os Middlewares são armazenados no arquivo `App/Http/Kernel`, na matriz `"$routeMiddleware"`. Como pode ver abaixo, eles possuem uma chave, e o seu valor é uma classe. Para utilizar um ou outro, a chave deve ser informada.

```
/**
 * The application's route middleware.
 *
 * These middleware may be assigned to groups or used individually.
 *
 * @var array
 */
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'password.confirm' => \Illuminate\Auth\Middleware\RequirePassword::class,
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
    'session_validate' => \App\Http\Middleware\SessionValidate::class, // Middleware criado para validar sessões
];
```

### 3.6.2. COMO UTILIZAR OS MIDDLEWARES PARA PROTEGER AS ROTAS

Uma vez que o middleware existe no kernel HTTP, você pode usar o **método middleware()** para ligar proteger a rota com um dos middlewares disponíveis.

```
use App\Http\Controllers\UserController;
```

```
Route::get('/user', [UserController::class, 'index']->name("user")->middleware("auth");
```

**Múltiplos podem ser utilizados para uma mesma rota**, e basta que suas chaves sejam informadas dentro de um array.

```
->middleware(["auth", "can"]);
```

Também existem os **grupos de middleware**, que é uma matriz de nome “\$middlewareGroups”, também existente no arquivo Kernel.php. Isso serve para que múltiplos middlewares comuns sejam utilizados a partir de uma só chave. Existem dois grupos: o “web” e o “api”.

```
Route::middleware("web")->group(function(){
```

```
// Configuração das rotas
```

```
// Serão protegidas pelo middleware “web”
```

```
}
```

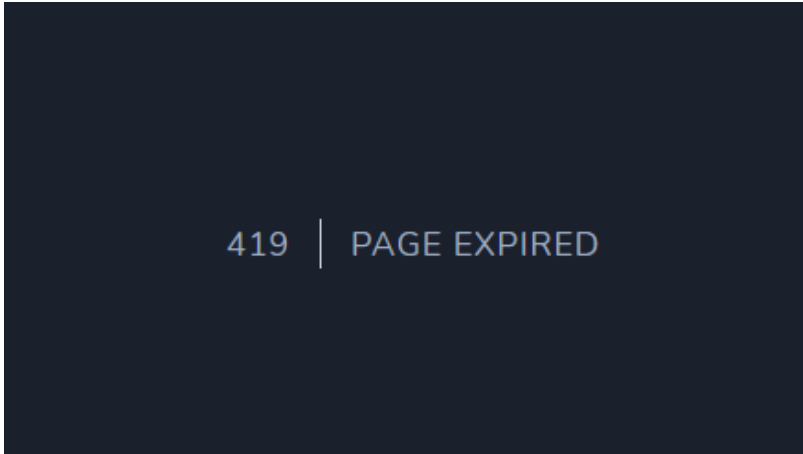
### 3.7. PROTEÇÃO CONTRA ATAQUES CSRF

As falsificações de solicitação entre sites são um tipo de exploração maliciosa em que comandos não autorizados são executados em nome de um usuário autenticado.

**O Laravel gera automaticamente um "token" CSRF para cada sessão de usuário ativa gerenciada pela aplicação.** Esse token é usado para verificar se o usuário autenticado é a pessoa que está realmente fazendo as solicitações ao aplicativo.

Sempre que definir um formulário HTML "POST", "PUT", "PATCH" ou "DELETE" em seu aplicativo, você deve incluir o token com a diretiva @csrf. Na verdade, se trata de um input oculto com o atributo “value” sendo igual a “csrf\_token()” – método que retorna o token atualmente ativo.

Se esse token não for incluído no formulário, isto irá ocorrer quando seus dados forem enviados:



419 | PAGE EXPIRED

Também, **já antecedendo uma dúvida**, embora essa diretiva necessite de uma sessão ativa, com um token que seja igual ao dela mesma, não é necessário criar uma especificamente para isso, porque **por padrão o Laravel já trabalha com uma sessão ativa**. Isso será abordado no tópico sobre sessões.

#### 4. OS CONTROLLERS

Como já foi explicado, os controladores servem para controlar as operações das rotas da aplicação. Um só controlador pode controlar uma rota, ou várias, com uma mesma action ou mais de uma.

É conveniente que seja utilizado um único controlador para uma única rota, ou métodos diferentes para diferentes rotas, devido ao nível de complexidade e densidade de um código de um controlador dedicado a resolver operações de múltiplas rotas.

##### 4.1. CRIAÇÃO DE UM CONTROLLER

A criação de um Controller se dá facilmente com a utilização do console Artisan. O comando é esse:

```
php artisan make:controller UserController
```

##### 4.2. CRIANDO UM CONTROLLER SINGLE ACTION

Um controller single action, ou de “ação única”, é um que resolve as operações de uma única rota. Esse tipo de controller não precisa que seu método seja especificado na configuração da rota que irá tratar.

Além disso ele possui uma sintaxe diferente do controller padrão. Ao invés de ter múltiplas actions, ele possui uma só e que é criada automaticamente: `__invoke()` é o seu nome.

```
php artisan make:controller UserController --invokable
```

### 4.3. CAPTURANDO REQUESTS

A classe **Illuminate\Http\Request** do Laravel fornece uma maneira orientada a objetos para interagir com a solicitação HTTP atual sendo tratada por sua aplicação, bem como recuperar a entrada, cookies e arquivos que foram enviados com a solicitação.

**O primeiro exemplo de caso de uso** seria em controladores que possuem métodos para tratar requisições de formulário. O que ocorre é que eles devem possuir um parâmetro `$request` do tipo “Request”. Ele é um objeto que recebe os dados da requisição enviada para a rota.

Por exemplo, se o formulário é de login, e possui dois inputs, um de senha, com name “senha” e outro de email, com name “email”, basta utilizar `$request->senha` e `$request->email` para acessar o dado enviado por cada um dos inputs.

**O segundo exemplo de caso de uso** seria para recuperar parâmetros da URL ou query strings. Por exemplo, se a rota atual é “/user?id=123”, para recuperar o valor da query string “id” basta escrever `$id = request(“id”)`.

Esses seriam alguns dos principais exemplos, mas existem incontáveis. O objeto `$request` permite recuperar o caminho da solicitação, com `$request->path()`, inspecionar a atual rota com `$request->is(“/nome_rota”)`, inspecionar a rota atual se for nomeada, com `$request->routeIs(“nome_rota”)`, dados dos headers, entre diversas outras coisas.

### 4.4. OS RETORNOS DOS CONTROLLERS

Todas as rotas e controladores devem retornar uma resposta a ser enviada de volta ao navegador do usuário. O Laravel oferece várias maneiras diferentes de retornar respostas.

#### 4.4.1. RETORNAR UMA VIEW

Os controllers se comunicam ou não com models, para capturar dados do banco de dados e nutrir as views que retornam para o usuário.

Para retornar essas views o controller utiliza o método `view(“nome_da_view”)` antecedido do termo reservado para retornos, “return”.

```
return view(“home”);
```

Também é possível enviar dados para as views; possibilidade essa que é necessária quando devem ser nutridas com dados do banco de dados.

```
return view(“home”, [“personName” => $personName]);
```



Os dados enviados para uma view, que utiliza o template blade, são utilizados a partir de variáveis php, existentes entre dois pares de chaves, e que possuem o exato mesmo nome da chave utilizada para referenciar o dado a ser enviado para a view, no método view(). Ou seja, neste caso seria:

```
{{ $personName }}
```

#### 4.4.2. RETORNAR UM OBJETO RESPONSE

Também é possível instâncias da classe **Illuminate\Http\Response**, que são objetos que organizam os retornos dos controllers quando não são views.

Retornar uma instância de Response permite que você personalize o conteúdo da resposta, o código de status HTTP e os cabeçalhos.

```
use Illuminate\Http\Response;

return response("Hello World", 200);

return response('Hello World', 200) ->header('Content-Type', 'text/plain');
```

#### 4.4.3. REDIRECIONAMENTO COM REDIRECT

Os redirecionamentos são instâncias da classe **Illuminate\Http\RedirectResponse**. Existem várias maneiras de gerar um redirecionamento. O método mais simples é usar o redirect global:

```
return redirect("home/dashboard"); // Rota comum

return redirect("dashboard"); // Rota nomeada
```

#### 4.4.4. TIPO DO DADO DA RESPONSE

O response pode ser usado para gerar outros tipos de instâncias de resposta também, como JSON, e até mesmo um caminho de arquivo para que seja baixado.

```
return response()->json(["name" => "Batman"]); // Retorna JSON

return response()->download($pathToFile); // Faz download

return response()->file($pathToFile); // Visualização de um arquivo
```

#### 4.5. CONTROLLERS DE RECURSOS: PARA CRUDS GENÉRICOS

O roteamento de recursos do Laravel permite atribuir as rotas típicas de criação, leitura, atualização e exclusão ("CRUD") para um controlador com uma única linha de código.

Para começar, podemos usar a opção `make:controller` do comando Artisan `--resource` para criar rapidamente um controlador para lidar com estas ações:

```
php artisan make:controller UserController --resource
```

O que ocorre é que ele é criado automaticamente, como também ocorre com controladores comuns, mas com **actions e rotas nomeadas prontamente disponíveis** para realização de ações CRUD.

```
Route::resource("user", "UserController");
```

Neste caso, da criação de um controlador de recursos `"UserController"` para a rota `"/user"`, o que ocorreria, por debaixo dos panos, seria, além da geração do próprio controlador, também seriam geradas actions para ele, pré-definidas, e todas as rotas para que sejam acessadas.

#### 4.5.1. A ORGANIZAÇÃO DE UM CONTROLADOR DE RECURSOS

Um controlador de recursos criado, com o Artisan, tem, por padrão, uma action `"index"`, que é chamada quando acessada a rota principal configurada na rota, e algumas outras para realizar operações CRUD genéricas, e que são acessadas em níveis superiores em relação à rota principal.

Por exemplo, sendo a rota principal `"/user"`, se o usuário acessar `"/user/create"` irá ativar a action nativa `"create"` do controlador de recursos, que é usada para retornar view de operação de inserção de dados no banco.

Um controlador de recursos tem actions para lidar com as views de formulário de operações CRUD, e outras actions para lidar com as requisições POST dessas operações. Para a action de formulário `"create"`, existe a action `"store"` para processar o SQL INSERT, para a action de formulário `"edit"`, existe a `"update"` para processar o SQL UPDATE.

Ou seja: para cada action de formulário existe uma action correspondente para o seu processamento.

#### 4.5.2. CONTROLLER RESOURCE COM MODEL ESPECIFICADO

É possível que os métodos do controlador de recursos trabalhem com uma instância de um model especificado na sua criação. Para isso basta utilizar você pode usar a opção `--model` na geração do controlador:

```
php artisan make:controller UserController --model=User --resource
```

#### 4.5.3. CONTROLLER RESOURCE PARA FORMULÁRIOS

É possível fornecer a opção `--requests` ao gerar um controlador de recursos para instruir o Artisan uma classe dedicada a solicitação de formulários:

```
php artisan make:controller UserController --model=User --resource --requests
```

#### 4.5.4. ESPECIFICANDO AS ACTIONS DISPONÍVEIS

As vezes uma rota necessita de um controlador de recursos, mas não de todas as suas actions, mas de algumas, como a index e outra.

Para especificar quais as actions disponíveis para uma determinada rota, basta configurar a rota seguindo esta lógica:

```
Route::resource('photos', PhotoController::class)->only(['index', 'show']); // Apenas
```

```
Route::resource('photos', PhotoController::class)->except(['create', 'store']); // Exceto
```

### 5. AS VIEWS

As visualizações, ou “views”, tem como função a separação da lógica do controlador, ou do aplicativo, da lógica de apresentação. Esses arquivos são armazenadas na pasta `resources/views`.

#### 5.1. O TEMPLATE BLADE

Você pode criar uma view colocando um arquivo com a extensão “.blade.php” no diretório `resources/views`. Essa extensão, “.blade.php”, informa ao framework que o arquivo contém um template Blade.

Blade é o mecanismo de modelagem simples, mas poderoso, que vem com o Laravel. Todos os modelos Blade são compilados em código PHP simples e armazenados em cache até serem modificados, o que significa que adiciona essencialmente zero sobrecarga ao aplicativo.

#### 5.2. ENVIANDO DADOS PARA AS VIEWS

Como ficou nítido em seções anteriores, é possível passar uma matriz de dados para nutrir views com valores, por exemplo, processados nos controladores.

```
return view('home', ['username' => $username]);
```

Sendo enviados dessa maneira, em forma de par chave-valor, os dados ficam disponíveis para serem utilizados nas views a partir de variáveis php com o exato mesmo nome da chave do dado.

Para acessar o dado do exemplo, “username”, na view teria que ser escrito `{{ $username }}`.

### 5.3. AS DIRETIVAS DO TEMPLATE BLADE

O Blade também fornece atalhos convenientes, cuja sintaxe utiliza o prefixo `@`, para estruturas de controle PHP comuns, como `require`, instruções condicionais e loops.

#### 5.3.1. DIRETIVA @EXTENDS

Essa diretiva serve como um mecanismo de herança de views. Ela permite que você “estenda” um modelo “parent” em outro, que se torna o “child”.

Esse recurso será citado novamente no tópico sobre Layouts, porque é para eles que é comumente utilizado.

#### 5.3.2. DIRETIVA @SECTION

Essa diretiva, como o nome indica, define uma seção de conteúdo. Na prática funciona como uma variável, ou um par chave-valor, e é criada em uma view “parent” para ser utilizada em uma view “child”.

```
@section("chave", "valor")
```

```
@section("chave")
```

```
// Conteúdo HTML
```

```
@endsection
```

#### 5.3.3. DIRETIVA @YIELD

Essa diretiva serve para que, na view “parent”, os valores das diretivas `@section`, da view “child”, sejam recuperadas.

```
@yield("chave")
```

#### 5.3.4. OUTRAS DIRETIVAS

Outros tipos de diretivas são as condicionais, de loop, de autenticação, de validação de formulários, entre muitas outras. Clique [aqui](#) para saber mais.

### 5.4. LAYOUTS E HERANÇA: VIEW CONSTANTES E VARIÁVEIS

É comum que aplicações web utilizem estruturas constantes em diversas páginas, como um mesmo header, e um mesmo footer, e apenas alterem conteúdos pontuais entre elas.

Assim, é interessante que essas estruturas constantes, ou que se repetem com frequência, possam ser codificadas em um único arquivo, e serem estendidas por outros. Pois bem: isso existe, e o nome desse tipo de arquivo é Layout.

Em termos de diretórios, o que pode ser feito é a criação de uma pasta de nome “Layouts”, em resources/views, que terá esse tipo de arquivo.

Já quanto à codificação, uma view de Layout deve ter essencialmente estruturas que se espera que irão repetir muitas vezes no aplicativo, e outras que irão receber conteúdos variáveis das suas views filhas, isto é, que irão estende-la com a diretiva @extends.

Assim, as views que variam, terão o conteúdo, por exemplo, de cada página, definidos dentro de um bloco @section, e irão estender o Layout com a diretiva @extends. Enquanto isso, logicamente, o próprio Layout deverá ter, obrigatoriamente, diretivas @yield para receber esses conteúdos a partir das suas chaves.

Veja abaixo um exemplo de Layout, em que @yield(“content”) recebe o conteúdo de @section(“content”) das views filhas:

```
<header class = "layout_header">
    <h1></h1>
</header>

<section class = "layout_section">
    @yield("content")
</section>

<footer class = "layout_footer">
    <p>Aprendendo Laravel &copy; 2021</p>
</footer>
```

## 6. SESSÕES

Como os aplicativos baseados em HTTP não têm estado, as sessões fornecem uma maneira de armazenar informações sobre o usuário em várias solicitações. Essas informações do usuário são normalmente colocadas em um armazenamento / back-end persistente que pode ser acessado a partir de solicitações subsequentes.

### 6.1. O APLICATIVO LARAVEL JÁ POSSUI UMA SESSÃO ATIVA

No ambiente do Laravel, mesmo que uma sessão não seja definida, já existe uma. Sim, um aplicativo Laravel tem, por padrão, uma sessão ativa. Para verificar os dados dessa sessão inicializada automaticamente, basta digitar o seguinte código:

```
Print_r($request->session->all());
```

Como é possível ver, os dados da sessão são recuperados a partir do objeto `$request` da classe **Illuminate\Http\Request**. Executando esse comando, **alguns dados serão mostrados**, como a URL atual, e o **valor do TOKEN** de segurança – sim, o mesmo utilizado na diretiva CSRF.

### 6.2. CRIANDO UM VALOR DE SESSÃO

Criar uma sessão consiste, na verdade, em **atribuir novos valores** a essa citada sessão já existente no aplicativo Laravel.

**Existem dois jeitos** de atribuir novos valores para a sessão. **O primeiro** seria utilizando o objeto `Request`, da classe **Illuminate\Http\Request**, com os métodos `->session()->put("chave", "valor")`. **O segundo** seria utilizar a classe `Facade Session`, **Illuminate\Support\Facades\Session**, escrevendo `Session::put("chave", "valor")`.

### 6.3. RESGATANDO VALOR DA SESSÃO

Existem aqui, também, duas maneiras principais de recuperar os dados de uma sessão: a partir da instância da classe `Request $request`, ou a partir da `Facade Session`.

#### 6.3.1. RECUPERANDO DADOS DA SESSION COM A INSTÂNCIA DE REQUEST

Com a instância de `Request` alguns métodos interessantes são possíveis:

```
$request->session()->get("chave"); // Um dado específico
```

```
$request->session()->all(); // Todos os dados
```

```
$request->session()->has('users'); // Verificação de chave com valor não nulo
```

```
$request->session()->missing('users'); // Verificação se é nulo ou se não existe
```

#### 6.3.2. RECUPERANDO DADOS DA SESSION COM O FACADE SESSION

Com o `Facade Session` o método seria: `session('key')`.

## 7. SISTEMA DE LOGS

Para ajudá-lo a aprender mais sobre o que está acontecendo dentro de sua aplicação, o Laravel fornece serviços de registro robustos que permitem que você registre mensagens em arquivos, o log de erros do sistema e até mesmo para o Slack para notificar toda a sua equipe.

**Um sistema de log é imprescindível para qualquer sistema.** Ele permite o acompanhamento e verificação dos processos do aplicativo.

Um exemplo prático seria um sistema com usuários, e com seção de suporte. Se um usuário não consegue acessar o sistema, e registra uma reclamação sobre isso, com o registro de logs é possível verificar se ele realmente entrou alguma vez no sistema, quando, e quantas vezes, antes de recorrer ao suporte, errou os seus dados de acesso.

### 7.1. COMO FUNCIONA O SISTEMA DE LOGS

**O registro do Laravel é baseado em "canais".** Cada canal, já existente, representa uma maneira específica de gravar informações de log. Por exemplo, o canal “single” grava arquivos de log em um único arquivo de log, enquanto o “slack” envia mensagens de log para o Slack. Poderia ser criado um para enviar para o Whatsapp.

**Os canais representam, objetivamente, a identidade do log utilizado.** Se você quiser utilizar um ou outro, informará a mensagem que quer registrar e em qual, sendo a escolha de qual log utilizar, o nome do canal do log que será utilizado.

#### 7.1.1. ARQUIVO DE CONFIGURAÇÃO DOS LOGS

Todas as opções de configuração para o comportamento de registro do aplicativo estão armazenadas no **arquivo de configuração config/logging.php**. Nele é encontrada uma **matriz** de nome “**channels**”, e dentro dela as configurações dos canais.

```
'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['single'],
        'ignore_exceptions' => false,
    ],

    'single' => [
        'driver' => 'single',
        'path' => storage_path('logs/laravel.log'),
        'level' => env('LOG_LEVEL', 'debug'),
    ],

    'daily' => [
        'driver' => 'daily',
        'path' => storage_path('logs/laravel.log'),
    ],
]
```

Por padrão, o Laravel usará o canal “**stack**” ao registrar as mensagens. Esse canal é **usado para agregar vários canais** de log em um só.

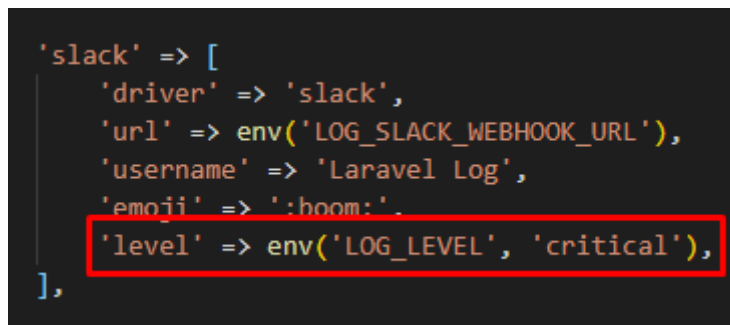
#### 7.1.2. DRIVERS DE CANAL DOS LOGS

Cada canal de log é alimentado por um “driver”. O driver determina como a mensagem de log é realmente gravada. Por exemplo, o driver “errorlog” registrará um log de erro.

Os drivers disponíveis podem ser encontrados [aqui](#).

#### 7.1.3. NÍVEL DE REGISTRO DO CANAL

Esta opção determina o “nível” mínimo que uma mensagem deve ter para ser registrada pelo canal.



```
'slack' => [
    'driver' => 'slack',
    'url' => env('LOG_SLACK_WEBHOOK_URL'),
    'username' => 'Laravel Log',
    'emoji' => ':boom:',
    'level' => env('LOG_LEVEL', 'critical'),
],
```

O exemplo acima é do canal Slack. Para que mensagens sejam gravadas nesse canal, o registro deve ser realizado com nível “critical”.

**Existem no total 8 níveis de registro:** emergency, alert, critical, error, warning, notice, info e debug.

#### 7.1.4. GRAVANDO MENSAGENS COM O FACADE LOG

E como gravar registros de log? Com a classe Illuminate\Support\Facades\Log. E conforme mencionado anteriormente, existem níveis de registro. Pois bem: quando registrado uma mensagem, é preciso informar o canal, o nível do registro e o seu conteúdo. No caso do canal slack poderia ser:

```
Log::channel('slack')->critical("The system crashed!");
```

#### 7.1.5. COMO CRIAR UM LOG/CANAL

Criar um log para o sistema, além dos já existentes, consiste em adicionar um canal no arquivo de configurações de logging. Um novo canal, assim como os já existentes, deverá ter um atributo “driver”, um “path”, que é o caminho do arquivo utilizado para armazenar as mensagens, e um “level”.



**FONTE**

<https://laravel.com/docs/8.x>