**Views mais flexíveis e poderosas**

**Views mais flexíveis e poderosas**

Um dos problemas de nosso código atual é que existe muita repetição em nossas views. Veja, por exemplo, como está nosso arquivo de listagem.php:

<html>

<head>

<link href="/css/app.css" rel="stylesheet">

<title>Controle de estoque</title>

</head>

<body>

<div class="container">

<h1>Listagem de produtos</h1>

<table class="table table-striped ...">

<!-- conteúdo omitido -->

</div>

</body>

</html>

Agora perceba a semelhança com o HTML do arquivo detalhes.php:

<html>

<head>

<link href="/css/app.css" rel="stylesheet">

<title>Controle de estoque</title>

</head>

<body>

<div class="container">

<h1>Detalhes do produto: <?= $p->nome ?> </h1>

<!-- conteúdo omitido -->

</div>

</body>

</html>

O conteúdo da *div* principal é diferente, afinal, cada página tem suas regras e objetivos específicos, mas todo o resto está igual! Com isso, no dia a dia se queremos criar uma nova página, sem pensar duas vezes acabamos copiando e colando alguma que existe, mudando apenas o miolo. Quem nunca fez isso?

O problema dessa abordagem é que, hora ou outra, você pode querer mudar o cabeçalho, menu, ou qualquer outro elemento que foi copiado e colado em quase todas as suas páginas. E agora? Replico a mudança nas minhas 200 views manualmente? Não parece uma solução ideal, não é?

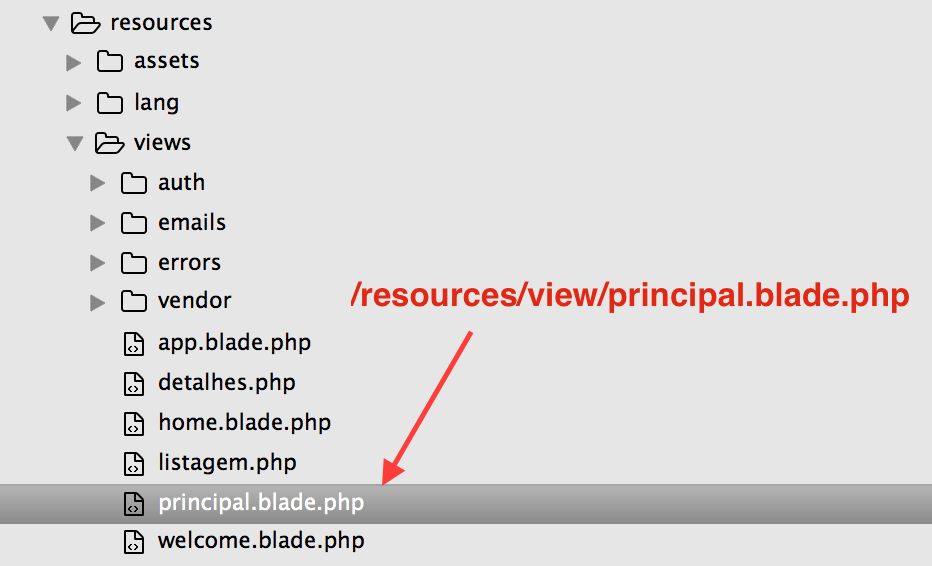
Nesta aula veremos como o Laravel pode nos ajudar nesse trabalho, tornando nossas views mais flexíveis, evitando repetições e ainda ganhando poderosos recursos que nos ajudam nas diferentes necessidades do nosso cotidiano.

**Blade, a template engine do Laravel**

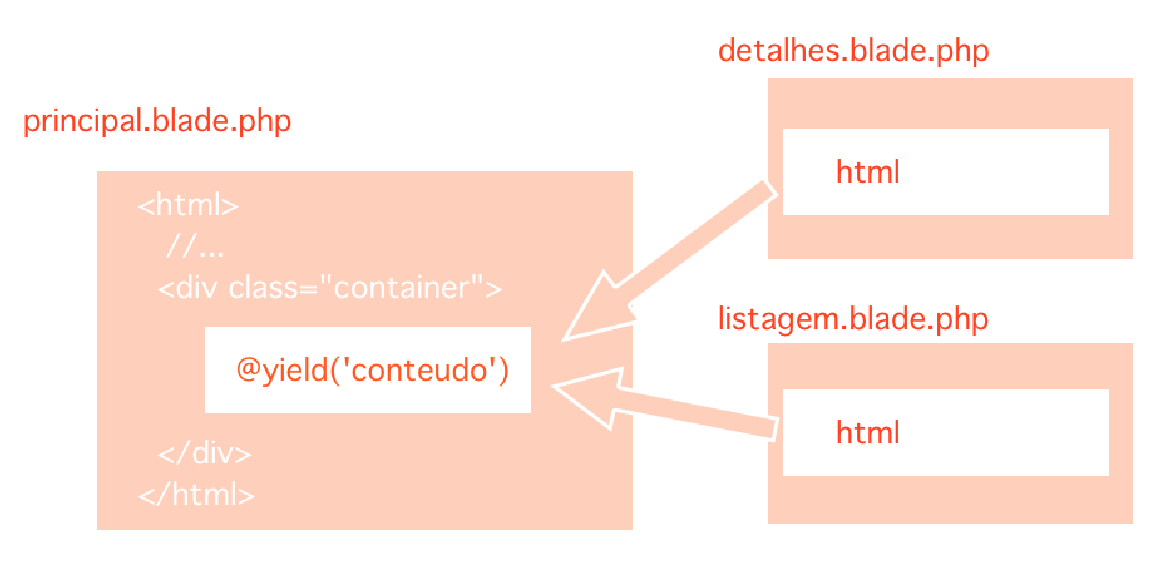
Se o problema estivesse acontecendo no código de nosso controller ou em nossas regras de negócio, bastaria extrair o código repetido pra um arquivo ou classe e pronto, poderíamos reutilizá-la sempre. Que tal se fosse possível fazer o mesmo na view, extraindo toda essa estrutura principal em um único arquivo e reutilizá-lo sempre que quiséssemos? Pois bem, isso não só é possível como também **muito recomendado**. Dessa forma, quando quisermos adicionar algum link, mudar os elementos ou fazer qualquer alteração nessa parte em comum, mudamos em um único lugar, e todo o resto é atualizado. O ganho em manutenibilidade é enorme.

Quem nos ajuda com esse trabalho é o **Blade**, a *template engine* do Laravel. Quer ver como é simples? Para resolver esse problema de repetição, basta criar um novo arquivo com o código em comum, que será nosso *layout* principal.

Um fator importante é que todo arquivo do blade precisar ter a extensão .blade.php, portanto, vamos chamá-lo de principal.blade.php, e lembre-se de que, como é uma *view*, precisa estar no diretório /resources/views.



Dentro desse arquivo, vamos colocar todo o HTML em comum, e utilizar a marcação @yield com o nome da seção que deverá ser sobrescrita pelas outras views. A ideia é simples, suas views "herdam" desse layout principal, e sobrescrevem as seções nele definidas:



O código do arquivo principal.blade.php pode ficar assim:

<html>

<head>

<link href="/css/app.css" rel="stylesheet">

<title>Controle de estoque</title>

</head>

<body>

<div class="container">

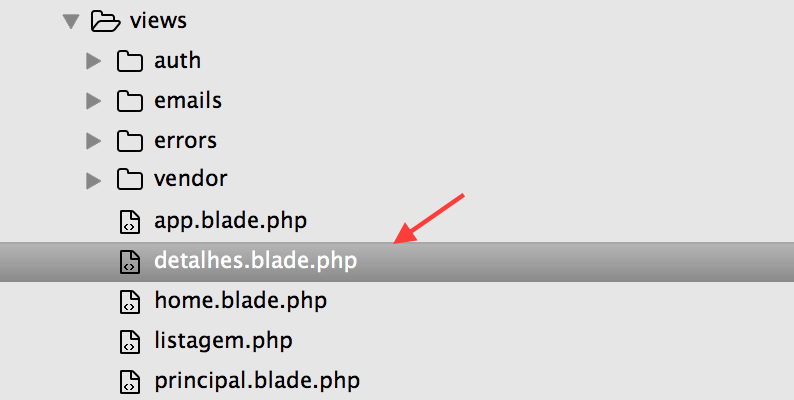
@yield('conteudo')

</div>

</body>

</html>

Nada muito diferente, não é? Tem todo o HTML em comum, e a marcação @yield('conteudo') para delimitar onde o conteúdo das outras views deve ser inserido. Vamos testar? O primeiro passo será mudar a extensão do arquivo detalhes.php para detalhes.blade.php.



Feito isso, podemos tirar todo o código que já existe no layout principal.blade.php, e deixar apenas o miolo que estava dentro da *div* com classe *container*. O arquivo vai ficar apenas com este conteúdo:

<h1>Detalhes do produto: <?= $p->nome ?> </h1>

<ul>

<li>

<b>Valor:</b> R$ <?= $p->valor ?>

</li>

<li>

<b>Descrição:</b> <?= $p->descricao ?>

</li>

<li>

<b>Quantidade em estoque:</b> <?= $p->quantidade ?>

</li>

</ul>

Ótimo! Mas ainda precisamos dizer que esse arquivo deve utilizar o layout principal, do arquivo principal.blade.php. Para isso, basta adicionar a marcação @extends com o nome do layout que ele deve herdar (neste caso: principal, que é o nome do arquivo sem a extensão).

@extends('principal')

<!-- restante do html aqui -->

Também é preciso colocar todo o conteúdo restante do nosso arquivo de detalhes dentro de uma @section, com o nome que foi definido no @yield do layout principal, que neste caso é conteudo:

@extends('principal')

@section('conteudo')

<h1>Detalhes do produto: <?= $p->nome ?> </h1>

<ul>

<li>

<b>Valor:</b> R$ <?= $p->valor ?>

</li>

<li>

<b>Descrição:</b> <?= $p->descricao ?>

</li>

<li>

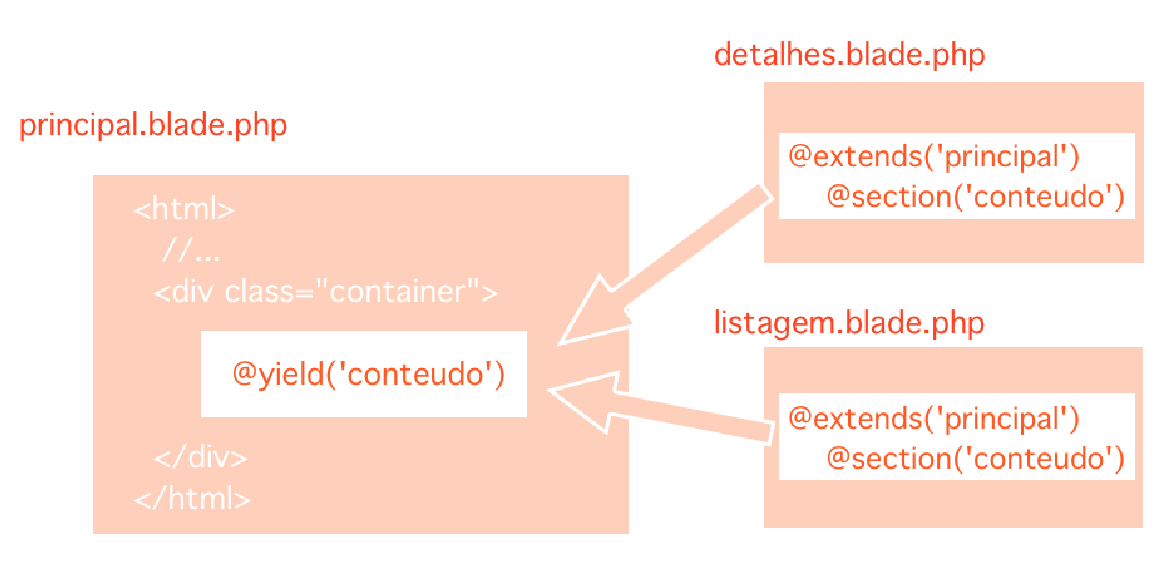
<b>Quantidade em estoque:</b> <?= $p->quantidade ?>

</li>

</ul>

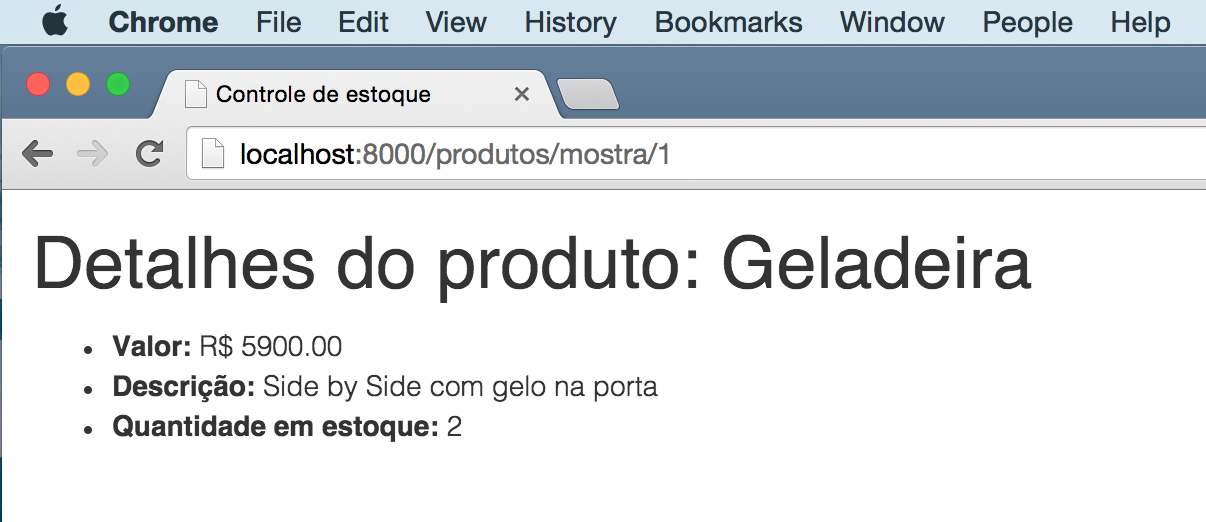
@stop

Veja que usamos um @stop para marcar o fim da section. Isso é necessário pois, em alguns momentos, podemos querer sobrescrever mais de uma section do layout principal.

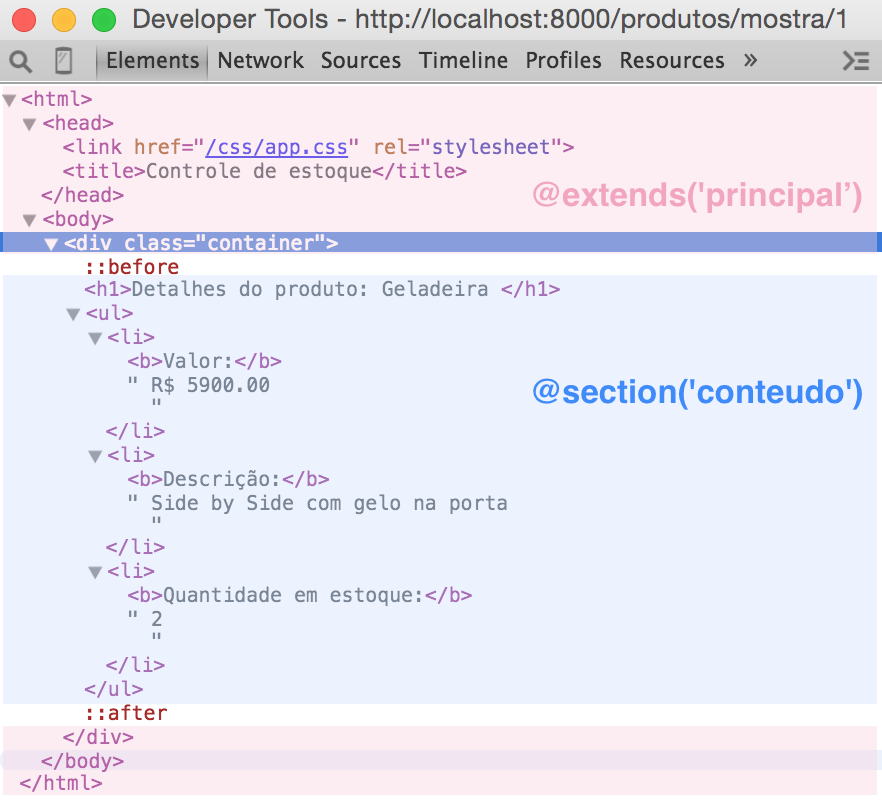


Vamos ver se após a mudança tudo continua funcionando? Basta acessar a página de detalhes de qualquer produto, por exemplo:

<http://localhost:8000/produtos/mostra/1>



Veja que, como esperado, a página continua funcionando normalmente. Se quiser, clique com o botão direito na página e selecione a opção *view page source*. Repare que o HTML continua o mesmo!



Legal, não é? Mas o **blade** nos oferece bem mais que isso! Temos muito mais o que explorar; vamos ver as formas de exibir o valor de nossas variáveis.

**Variáveis com blade**

Atualmente estamos usando a marcação do PHP puro para imprimir os valores do produto. Quando fazemos:

<li>

<b>Valor:</b> R$ <?= $p->valor ?>

</li>

Não há problema nisso e, se quiser, você ainda pode continuar exibindo dessa forma. Mas se quiser - e eu recomendo -, você também pode fazer da forma do blade, envolvendo a variável em chaves duplas como a seguir:

<li>

<b>Valor:</b> R$ {{ $p->valor }}>

</li>

A mudança é simples e o resultado será o mesmo, mas a sintaxe, em minha opinião, é um pouco mais agradável. Entretanto, o ganho vai além de uma sintaxe melhor, ou um caractere a menos. Você ganha bastante flexibilidade e alguns recursos extras, como o or, para definir um valor *default* caso a variável não esteja populada. Um exemplo seria:

{{$p->descricao or 'nenhuma descrição informada'}}

Legal, não é? Vamos conhecer diversos outros recursos no decorrer desta aula e curso, mas por enquanto, já podemos fazer a mudança no restante do arquivo, que deve ficar assim:

@extends('principal')

@section('conteudo')

<h1>Detalhes do produto: {{$p->nome}} </h1>

<ul>

<li>

<b>Valor:</b> R$ {{$p->valor}}

</li>

<li>

<b>Descrição:</b> {{$p->descricao}}

</li>

<li>

<b>Quantidade em estoque:</b> {{$p->quantidade}}

</li>

</ul>

@stop

**Migrando a listagem para Blade**

Vamos fazer a mesma alteração no arquivo listagem.php? Os passos serão os mesmos, começando por renomear o arquivo, que agora deve se chamar listagem.blade.php. Além disso, já podemos tirar todo o código que já está definido no layout principal e adicionar o @extends e @section. A listagem deve ficar assim:

@extends('principal')

@section('conteudo')

<h1>Listagem de produtos</h1>

<table class="table table-striped table-bordered table-hover">

<?php foreach ($produtos as $p): ?>

<tr>

<td><?= $p->nome ?></td>

<td><?= $p->valor ?></td>

<td><?= $p->descricao ?></td>

<td><?= $p->quantidade ?></td>

<td>

<a href="/produtos/mostra/<?= $p->id ?>">

<span class="glyphicon glyphicon-search"></span>

</a>

</td>

</tr>

<?php endforeach ?>

</table>

@stop

Legal, não é? O arquivo já fica um pouco mais limpo. Já podemos também mudar a forma como estamos exibindo nossas variáveis, utilizando as chaves duplas do blade. O corpo da tabela ficará assim:

<tr>

<td>{{ $p->nome }} </td>

<td>{{ $p->valor }} </td>

<td>{{ $p->descricao }} </td>

<td>{{ $p->quantidade }} </td>

<td>

<a href="/produtos/mostra/{{$p->id}}>">

<span class="glyphicon glyphicon-search"></span>

</a>

</td>

</tr>

**Loopings e mais loopings**

O blade também nos oferece uma maneira bastante interessante de enxugar a sintaxe do nosso foreach, na listagem de produtos. Veja que atualmente estamos fazendo com PHP puro, ou seja, utilizando a tag <?php ?>. Isso funciona, e você pode continuar fazendo assim se preferir, claro, mas que tal tirar toda essa verbosidade e adicionar um simples @ antes desse operador? Em outras palavras, em vez de fazer:

<?php foreach ($produtos as $p): ?>

<!-- codigo omitido -->

<?php endforeach ?>

Você pode mudar para:

@foreach ($produtos as $p)

<!-- codigo omitido -->

@endforeach

Isso melhora bastante a legibilidade do código. Agora o arquivo listagem.blade.php completo deve estar parecido com:

@extends('principal')

@section('conteudo')

<h1>Listagem de produtos</h1>

<table class="table table-striped table-bordered table-hover">

@foreach ($produtos as $p)

<tr>

<td> {{$p->nome}} </td>

<td> {{$p->valor}} </td>

<td> {{$p->descricao}} </td>

<td> {{$p->quantidade}} </td>

<td>

<a href="/produtos/mostra/{{$p->id}}>">

<span class="glyphicon glyphicon-search"></span>

</a>

</td>

</tr>

@endforeach

</table>

@stop

**Outros tipos de looping**

Além do foreach, também podemos fazer o mesmo com o for tradicional:

@for ($i = 0; $i < 10; $i++)

O indice atual é {{ $i }}

@endfor

Ou ainda com um while:

@while (true)

Entrando em looping infinito!

@endwhile

Ha ainda uma variação bastante interessante, chamada forelse. Se a lista for vazia, ele executa o código do bloco marcado com @empty:

@forelse($produtos as $p)

<li>{{ $p->nome }}</li>

@empty

<p>Não tem nenhum produto!</p>

@endforelse

Legal, não é? Mas como estamos fazendo o foreach no miolo de uma tabela, não seria muito interessante. No lugar disso, podemos usar um if!

**Adicionando condições na view**

Podemos fazer um if verificando se a lista de produtos está vazia, e se estiver, mostrar uma mensagem de que não existe nenhum produto cadastrado. Em PHP puro, o código ficaria assim:

<?php if(empty($produtos)) { ?>

<div class="alert alert-danger">

Você não tem nenhum produto cadastrado.

</div>

<?php } else { ?>

<h1>Listagem de produtos</h1>

<table>

<? foreach ($produtos as $p) { ?>

?>

<!-- continuação do código -->

<?php } ?> <!-- fechando o foreach -->

<?php } ?> <!-- fechando o else -->

Um pouco confuso e difícil de ler. Que tal com blade?

@if(empty($produtos))

<div class="alert alert-danger">

Você não tem nenhum produto cadastrado.

</div>

@else

<h1>Listagem de produtos</h1>

<table>

@foreach ($produtos as $p)

<!-- continuação do código -->

@endforeach

@endif

A ausência das chaves e tags do PHP ajuda bastante, nosso código fica mais simples e legível. Vamos fazer a alteração? O arquivo listagem.blade.php completo pode ficar assim:

@extends('principal')

@section('conteudo')

@if(empty($produtos))

<div>Você não tem nenhum produto cadastrado.</div>

@else

<h1>Listagem de produtos</h1>

<table class="table table-striped table-bordered table-hover">

@foreach ($produtos as $p)

<tr>

<td> {{$p->nome}} </td>

<td> {{$p->valor}} </td>

<td> {{$p->descricao}} </td>

<td> {{$p->quantidade}} </td>

<td>

<a href="/produtos/mostra/{{$p->id}}>">

<span class="glyphicon glyphicon-search"></span>

</a>

</td>

</tr>

@endforeach

</table>

@endif

@stop

Vamos testar? Para não ter que apagar os produtos do banco, vamos roubar um pouco, incluindo um array vazio no método lista do nosso controller:

class ProdutoController extends Controller {

public function lista(){

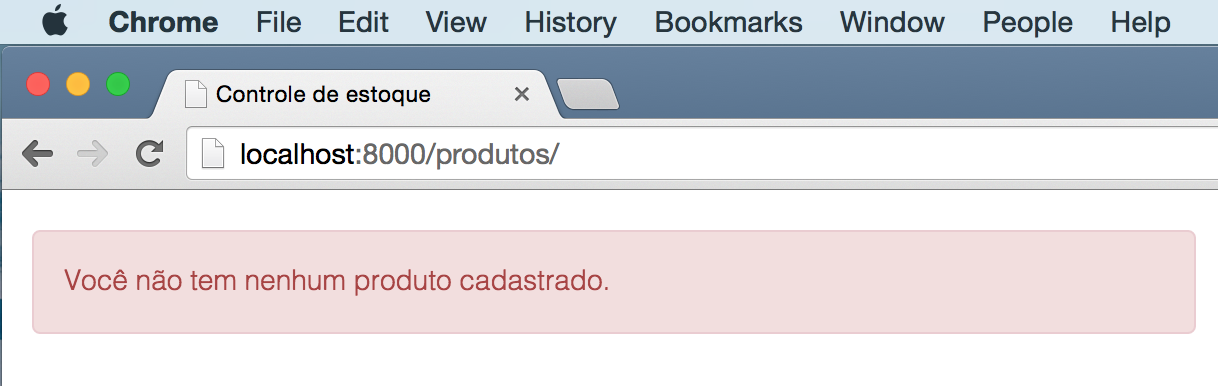
// busca os produtos do banco

return view('listagem')->with('produtos', array());

}

// continuação do código omitido

Com isso, ao acessar a listagem, o resultado será:



Excelente, tudo está funcionando como esperamos! Agora que testamos, não se esqueça de voltar o método lista ao normal, passando o array de $produtos no lugar de um array vazio.

**Outros formas de condicionar**

Além do @if e @else, você também pode usar @elseifs ou mesmo o @unless, que faz a condição inversa. Por exemplo, no caso a seguir, o valor condicionado sempre será exibido:

@unless (1 == 2)

Esse texto sempre será exibido!

@endunless

**Marcando produtos em falta no estoque**

Queremos agora pintar de vermelho as colunas da tabela que contêm produtos com quantidade menor ou igual a 1. Essa marcação visual vai nos ajudar a perceber quando um produto está em falta em nosso estoque.

Pra colorir a linha, podemos adicionar a classe danger, do bootstrap, nos trs dos produtos em falta. Apenas para ver o efeito da alteração, vamos adicionar a classe em todas as colunas:

<table class="table table-striped ...">

@foreach ($produtos as $p)

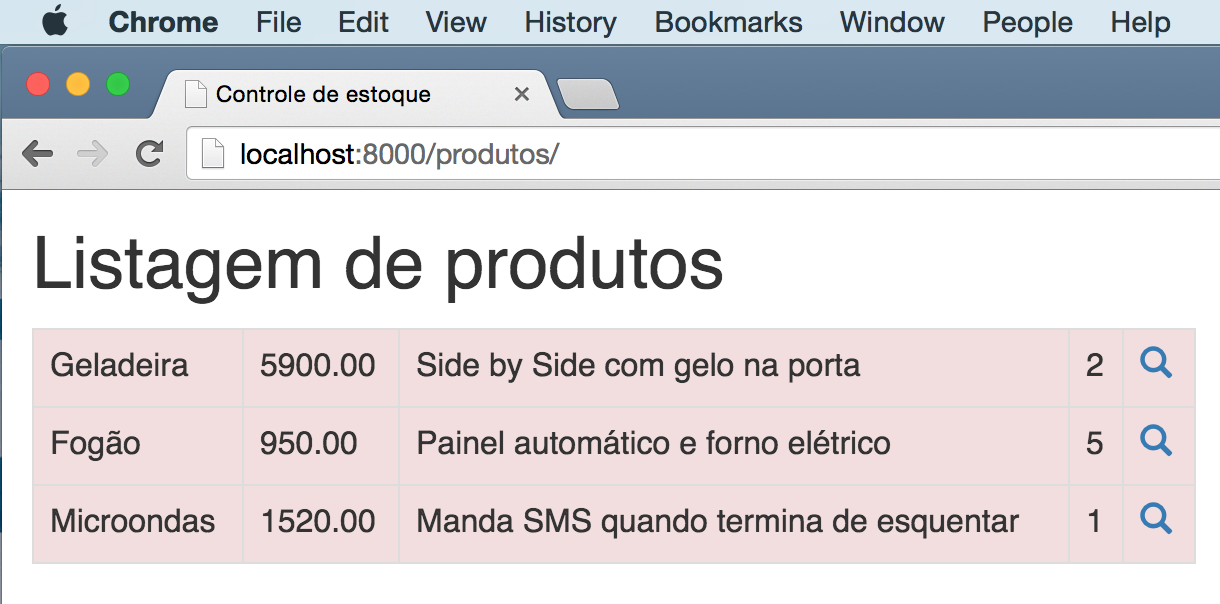
<tr class="danger">

<!-- código omitido -->

</tr>

@endforeach

Atualize a página da listagem para ver a diferença, ela agora deve estar parecida com:



Ótimo, agora só falta condicionar quais os produtos que devem ficar pintados de vermelho. Claro, um @ifresolveria, mas ficaria bem feio e verboso. No lugar disso, podemos fazer uma condição parecida com:

{{ $p->quantidade <= 1 ? 'danger' : ''}}

Esse é o tão conhecido ternário que, como você já deve ter percebido, tem a seguinte estrutura:

{{ condicao ? 'valor\_se\_true' : 'valor\_se\_false'}}

Ou seja, se a condição booleana for verdadeira (*true*), o primeiro valor será devolvido. Caso contrário, retornará o segundo valor. É importante saber que esse não é um recurso exclusivo do blade, claro, em PHP puro você também pode fazer:

<?php $p->quantidade <= 1 ? 'danger' : '' ?>

O que mudou foi a tag, assim como nos demais exemplos, mas nesse caso, usando o blade não precisamos adicionar nenhum @ para que isso funcione, basta colocar o ternário dentro das chaves duplas e pronto. Vamos testar? Basta adicionar essa condição dentro da class do tr. A tabela vai ficar assim:

<table class="table table-striped ...">

@foreach ($produtos as $p)

<tr class="{{$p->quantidade<=1 ? 'danger' : '' }}">

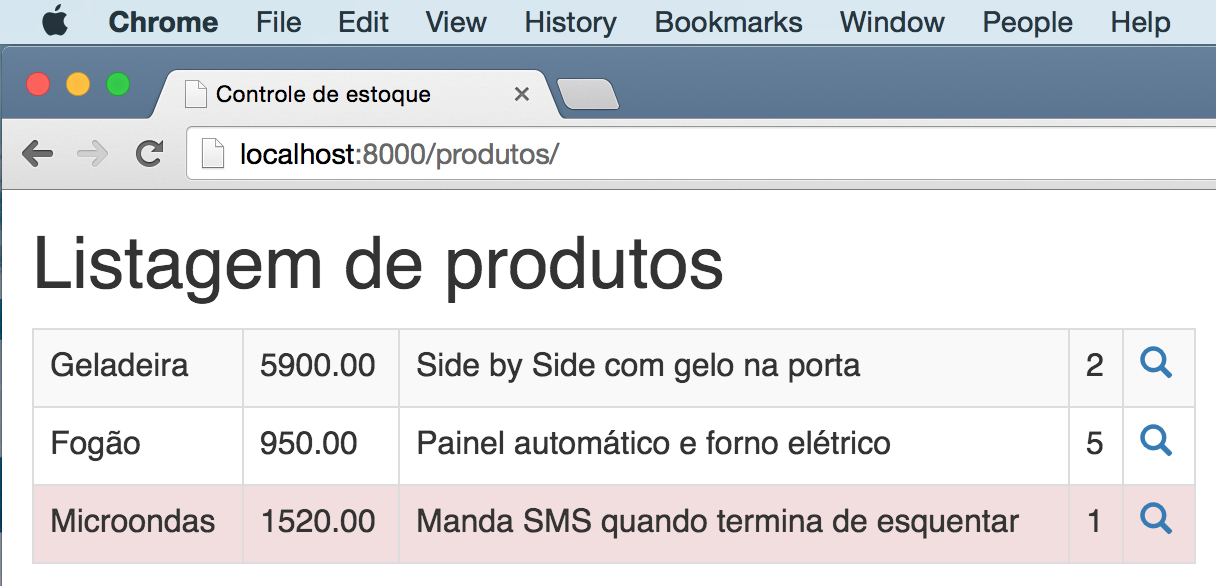
<!-- código omitido -->

</tr>

@endforeach

</table>

Pronto para testar? É só acessar a listagem novamente! No meu caso, ela ficará assim:



Também podemos adicionar uma legenda para deixar essa informação mais intuitiva. Vamos colocar um *span* logo depois do @endif, no final da tabela:

<h4>

<span class="label label-danger pull-right">

Um ou menos itens no estoque

</span>

</h4>

Veja que ele já tem algumas classes do bootstrap para estilizá-lo. Com essa alteração, o arquivo listagem.blade.php completo deve ficar assim:

@extends('principal')

@section('conteudo')

@if(empty($produtos))

<div class="alert alert-danger">

Você não tem nenhum produto cadastrado.

</div>

@else

<h1>Listagem de produtos</h1>

<table class="table table-striped table-bordered table-hover">

@foreach ($produtos as $p)

<tr class="{{$p->quantidade<=1 ? 'danger' : '' }}">

<td> {{$p->nome}} </td>

<td> {{$p->valor}} </td>

<td> {{$p->descricao}} </td>

<td> {{$p->quantidade}} </td>

<td>

<a href="/produtos/mostra/{{$p->id}}>">

<span class="glyphicon glyphicon-search"></span>

</a>

</td>

</tr>

@endforeach

</table>

@endif

<h4>

<span class="label label-danger pull-right">

Um ou menos itens no estoque

</span>

</h4>

@stop

Acesse a listagem novamente para ver o resultado! Ela deve estar parecida com:

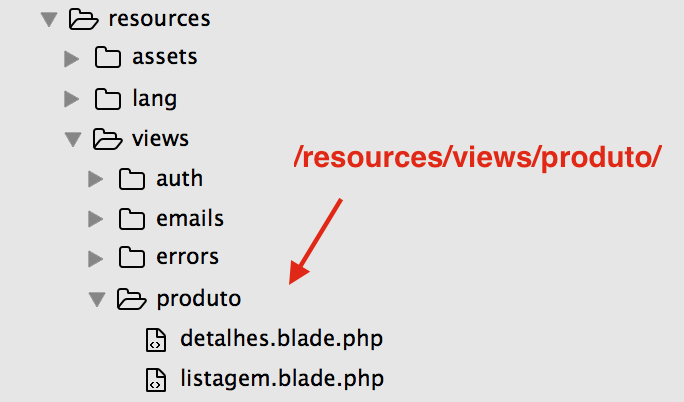


**Organizando nossas views**

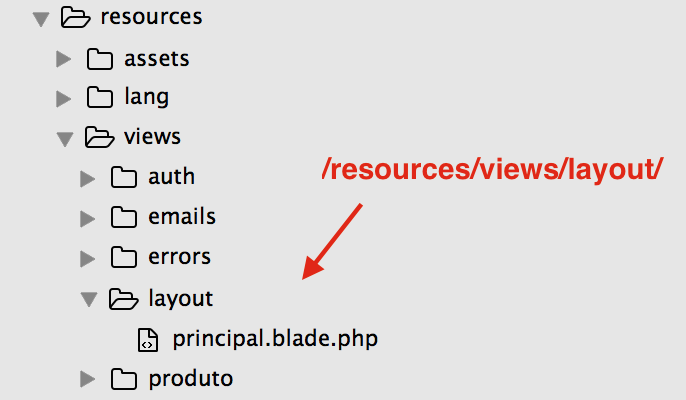
Conforme nosso sistema vai crescendo, mais e mais views devem aparecer e precisamos organizá-las de alguma forma. É muito comum separar esses arquivos por controller, ou qualquer estrutura que facilite seu trabalho e também o trabalho de sua equipe.

Um exemplo de separação por controller seria, se temos um ProdutoController, criar um diretório chamado produto e colocar nele todas as **views** relacionadas. O mesmo para usuários e qualquer outro elemento que venha a aparecer em nossa aplicação.

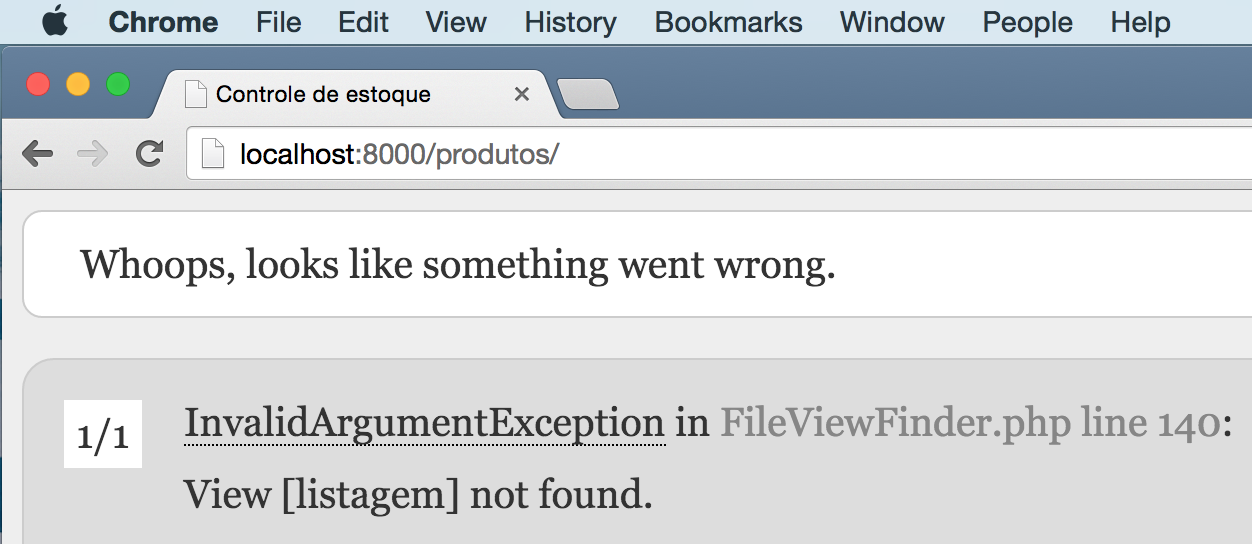
Vamos criar uma nova pasta chamada produto, dentro de resources/views, e mover para ela os arquivos listagem.blade.php e detalhes.blade.php. Seu projeto deve ficar assim:



Além disso, vamos criar uma nova pasta chamada layout, também em resources/views, para mover o arquivo principal.blade.php:



Mas, agora que mudamos isso, tente acessar qualquer página da sua aplicação. O resultado, talvez já esperado, será um erro!



A mensagem já dá uma pista: **View listagem not found**. Agora que mudamos a view para um subdiretório, precisamos alterar o retorno dos métodos do nosso controller. Atualmente fazemos algo como:

return view('listagem');

No lugar disso, precisamos ensinar ao Laravel que a view está dentro da pasta produto:

return view('produto/listagem');

Isso já funciona, mas é ainda mais comum fazermos desta forma:

return view('produto.listagem');

Veja que usamos um ponto no lugar da barra. O resultado será o mesmo, você pode escolher o que preferir. Nosso controller deve ficar assim:

class ProdutoController extends Controller {

public function lista(){

// código omitido

return view('produto.listagem')->with('produtos', $produtos);

}

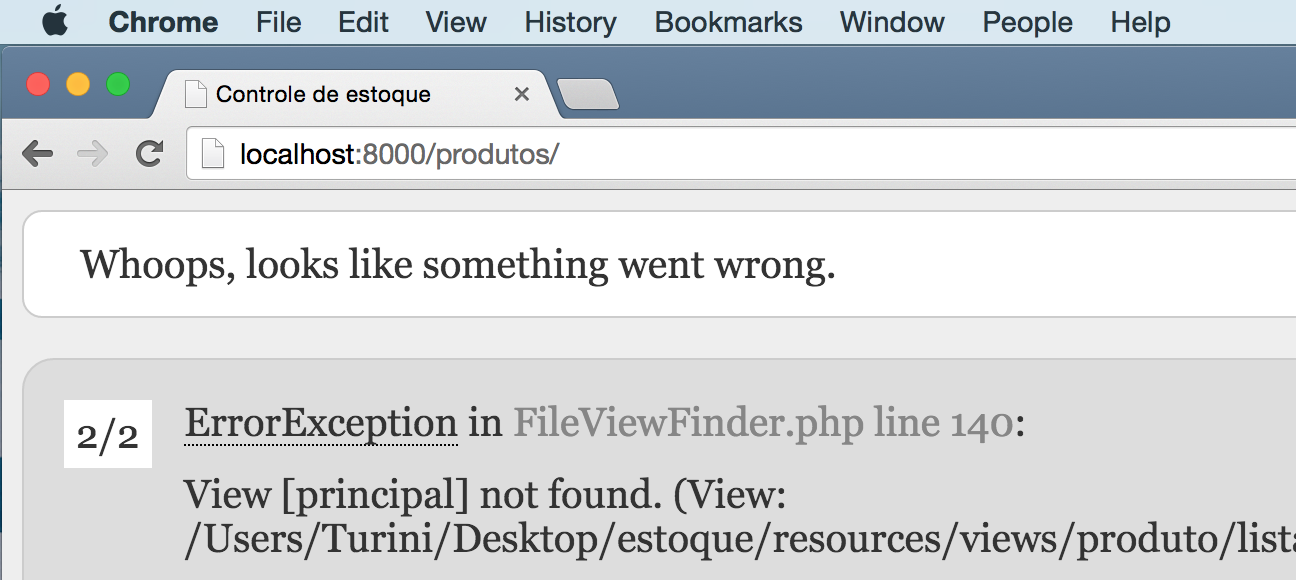
public function mostra($id){

// código omitido

return view('produto.detalhes')->with('p', $produto);

}

Vamos testar? Acessamos a página novamente e... ops!



O arquivo principal.blade.php não foi encontrado nas views, também precisamos mudar o lugar onde usamos o @extends! Agora, no lugar de @extends('principal'), precisamos mudar para @extends('layout.principal'). A alteração deve ser feita nos arquivos detalhes.blade.php e listagem.blade.php:

@extends('layout.principal')

@section('conteudo')

<!-- restante do conteúdo omitido -->

Ótimo, com isso nossa aplicação volta a funcionar como esperado.

**Estilizando o template principal**

Como nossas views seguem um template padrão, definido no principal.blade.php, agora fica muito mais fácil evoluir e estilizar o design da nossa aplicação web. Podemos adicionar um header com o nome do projeto, links, footer com mensagem de direitos autorais etc.

Veja como é simples. Como o layout está isolado, só precisamos aplicar as alterações no principal.blade.php e ela será propagada em todas as páginas. Um exemplo de layout seria:

<html>

<head>

<link href="/css/app.css" rel="stylesheet">

<link href="/css/custom.css" rel="stylesheet">

<title>Controle de estoque</title>

</head>

<body>

<div class="container">

<nav class="navbar navbar-default">

<div class="container-fluid">

<div class="navbar-header">

<a class="navbar-brand" href="/produtos">Estoque Laravel</a>

</div>

<ul class="nav navbar-nav navbar-right">

<li><a href="/produtos">Listagem</a></li>

</ul>

</div>

</nav>

@yield('conteudo')

<footer class="footer">

<p>© Livro de Laravel do Alura.</p>

</footer>

</div>

</body>

</html>

Repare que estamos importando um arquivo custom.css, que ainda não existe. Se quiser, você pode criar o arquivo dentro de /public/css/, com o conteúdo a seguir:

nav.navbar.navbar-default{

background-color: #337ab7;

border-color: #2e6da4;

margin-top: 10px;

}

nav.navbar.navbar-default a {

color: white;

}

li p.navbar-text, a.navbar-brand {

color: white!important;

font-weight: bolder;

}

footer.footer {

background-color: #337ab7;

border-color: #2e6da4;

margin-top: 55px;

color: white;

border-radius: 5px;

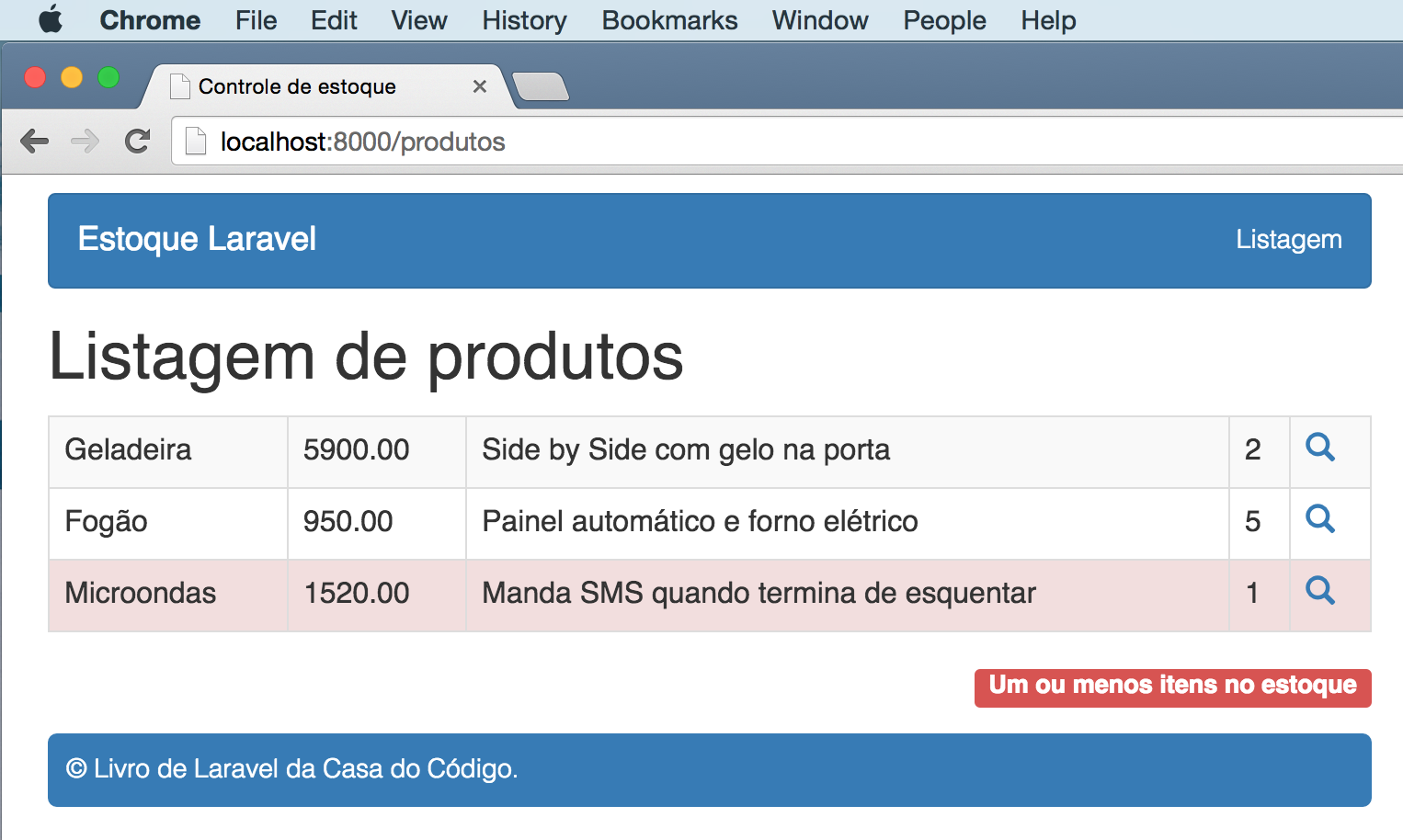
}

footer.footer p {

padding: 10px;

}

Com esse estilo, a listagem vai ficar parecida com:



E a view de detalhes deve ficar assim:

