1 Resumo Laravel

O que é o Laravel?

Laravel é um framework PHP gratuito e de código aberto, utilizado no desenvolvimento de sistemas para web. Algumas de suas principais características são permitir a escrita de um código mais simples e legível, e suporte a recursos avançados que agilizam o processo de desenvolvimento.

Principais recursos

O Laravel é baseado na arquitetura MVC (acrônimo para Model-View-Controller, ou Modelo-Visão-Controle, em português). MVC é um padrão de arquitetura de software focado em reuso de código, no qual ocorre a divisão da estrutura lógica de um sistema em 3 camadas: a do Modelo, relacionada ao banco de dados; a de Visão, vinculada a visualização dos dados e das páginas; e a do Controle, responsável pela conexão e transmissão de informações entre as camadas Modelo e Visão.

Além disso, o Laravel tem como principais recursos a utilização de um sistema modular para gerenciamento de dependências, diferentes formas de conexão e acesso a banco de dados relacionais, um motor próprio de templates para criação de interfaces, além de vários programas e serviços criados para facilitar a publicação e manutenção de sistemas criados com o framework.

1.1 Camada Model

O Laravel inclui o Eloquent, um mapeador objeto-relacional (ORM) que facilita a interação com seu banco de dados. Ao usar o Eloquent, cada tabela de banco de dados tem um "Modelo" correspondente que é usado para interagir com essa tabela. Além de recuperar registros da tabela do banco de dados, os modelos do Eloquent também permitem inserir, atualizar e excluir registros da tabela.

Para criar um Model executamos o comando:

- php artisan make:model <**nome_do_model**> -m

O parâmetro -m fará com que seja criado o arquivo de *migration*. Neste arquivo, indicaremos a estrutura desejada para a tabela no banco de dados: atributos, chave primária, chaves estrangeiras (campo para estabelecer o relacionamento entre tabelas)

1.1.1 Convenções

Por padrão. criamos cada Model com nome no singular e iniciando por maiúscula, e o Laravel criará a "migration" no plural em minúsculas.

Exemplo:

- php artisan make:model Pessoa -m

O arquivo da classe model será nomeado Pessoa.php e o arquivo de migrations como pessoas.

1.1.2 Eloquent - Relacionamentos

As tabelas de banco de dados costumam estar relacionadas umas às outras. Por exemplo, uma postagem de blog pode ter muitos comentários ou um pedido pode estar relacionado ao usuário que o fez. O Eloquent facilita o gerenciamento e o trabalho com esses relacionamentos e oferece suporte a

uma variedade de relacionamentos comuns:

Os relacionamentos do Eloquent são definidos como métodos em suas classes de modelo do Eloquent. Como os relacionamentos também servem como construtores de consulta poderosos , definir relacionamentos como métodos fornece recursos poderosos de encadeamento e consulta de métodos.

Um para muitos

Um relacionamento um para muitos é usado para definir relacionamentos em que um único modelo é o pai de um ou mais modelos filhos. Por exemplo, uma postagem de blog pode ter um número infinito de comentários. Como todos os outros relacionamentos do Eloquent, os relacionamentos de um para muitos são definidos pela definição de um método em seu modelo do Eloquent:

Como exemplo, faremos um relacionamento entre Autor e Livros. Temos então que **Um Autor** pode ter **Muitos livros**:

app\Models\Autor.php

```
1 public function livro(){
2   return $this->hasMany('App\Models\Livro');
3 }
```

Lembre-se de que o Eloquent determinará automaticamente a coluna de chave estrangeira apropriada para o modelo. Por convenção, o Eloquent pegará o nome do modelo pai e o sufixará com _id. Portanto, neste exemplo, o Eloquent assumirá que a coluna de chave estrangeira no modelo **Livro** é autor_id.

Caso não se tenha seguido o padrão, o campo "chave estrangeira" deverá ser informado como parâmetro:

```
1 public function book(){
2   return $this->hasMany('App\Models\Livro', 'foreign_key', 'local_key');
3 }
```

Um para muitos (inverso) / pertence a

Agora que podemos acessar todos os **Livros** de um **Autor**, vamos definir um relacionamento para permitir que um Livro acesse sua postagem pai. Para definir o inverso de um relacionamento hasMany, defina um método de relacionamento no modelo filho que chama o método belongsTo:

app\Models\Livro.php

```
1 public function autor(){
2   return $this->belongsTo('App\Models\Autor');
3 }
```

Caso não se tenha seguido o padrão para definição das chaves primária e/ou estrangeira, deverá ser informado como parâmetro:

```
1 public function book(){
2   return $this->belongsTo('App\Models\Autor', 'foreign_key', 'owner_key');
3 }
```

1.1.3 Eloquent - Mutators

Mutators são métodos definidos para acessar e/ou para enviar dados para o BD.

São definidos então como: Acessadores e Mutadores.

Definindo Acessadores

Um acessador transforma um valor de atributo do Eloquent quando ele é acessado no BD. Para definir um acessador, crie um método **getNomedoCampoAttribute** em seu modelo onde Nomedo-Campoé o nome da coluna que você deseja acessar.

Exemplo

Vamos definir então acessadores para os campos **dataNascimento** do nosso **Autor** e para **data-Lancamento** do nosso **Livro**

app\Models\Autor.php

```
1 public function getDataNascimentoAttribute(){
2   $dataConvertida = implode('/', array_reverse(explode('-', $this->
        attributes['dataNascimento'])));
3   return $dataConvertida;
4 }
```

app\Models\Livro.php

```
1 public function getDataLancamentoAttribute(){
2   $dataConvertida = implode('/', array_reverse(explode('-', $this->
        attributes['dataLancamento'])));
3   return $dataConvertida;
4 }
```

Como você pode ver, o valor original da coluna é utilizado pelo acessador, permitindo que você manipule e retorne o valor com as conversões que forem necessárias.

Definindo Mutadores

Um modificador transforma um valor de atributo do Eloquent quando ele é definido. Para definir um modificador, defina um método **setNomedoCampoAttribute** em seu modelo onde Nomedo-Campo é o nome da coluna que você deseja acessar.

Exemplo

Vamos definir então Mutadores para os campos **dataNascimento** do nosso **Autor** e para **data-Lancamento** do nosso **Livro**

app\Models\Autor.php

```
1 public function setDataNascimentoAttribute($value){
2   $this->attributes['dataNascimento'] = implode('-', array_reverse(explode('
        /', $value)));
3 }
```

app\Models\Livro.php

```
public function setDataLancamentoAttribute($value){
  $this->attributes['dataLancamento'] = implode('-', array_reverse(explode('
      /', $value)));
}
```

Como você pode ver, o valor é recebido pelo modificador, permitindo que você manipule e envie para o BD o valor com as conversões que forem necessárias.

1.2 Camada Controller

O controller faz parte do design pattern MVC. A função dos Controllers é organizar e agrupar requests relacionadas, manipulando sua lógica em uma única classe.

O controller é onde manipulamos a lógica das Requests, recebendo os dados do model e transmitindo-os para a view. O controller abstrai a complexidade da rota que, como já diz o nome, apenas roteará a Request feita para sua devida lógica.

Por padrão, os controladores são armazenados no diretório app/Http/Controllers.

1.3 Criando um Controlador

Criarmos um controlador através do artisan, com o comando:

- php artisan make:controller <nome_do_Controller>

Se desejarmos que a estrutura básica para um CRUD seja criada automaticamente:

- php artisan make:controller <nome_do_Controller> - -resource
- OU
- php artisan make:controller <nome_do_Controller> -r

O controller sempre estende a classe Controller padrão do Laravel. Assim, você já terá convenientemente métodos comuns como, por exemplo, o middleware.

Para todo CRUD básico, nós teremos até sete funções que resolverão em sete rotas diferentes.

- index - Lista os dados da tabela
- show - Mostra um item específico
- create - Retorna a View para criar um item da tabela
- store - Salva o novo item na tabela
- edit - Retorna a View para edição do dado
- update - Salva a atualização do dado
- destroy - Remove o dado

1.4 Camada View

O Laravel utiliza um sistema de template chamado Blade, que se diferencia de outras soluções PHP por não se restringir ao uso dessa linguagem em suas páginas. Além disso, no Blade, cada view é compilada e armazenada em cache até sofrer alguma alteração, deixando assim seus templates mais leves.

Você pode criar uma visualização colocando um arquivo com a extensão .blade.php no diretório resources/views do seu aplicativo . A extensão .blade.php informa ao framework que o arquivo contém um template Blade . Os modelos Blade contêm HTML e também diretivas Blade que permitem ecoar facilmente valores, criar instruções "if", iterar dados e muito mais.

Otimizando Visualizações

Por padrão, as visualizações do modelo Blade são compiladas sob demanda. Quando um pedido é executado para renderizar uma visão, o Laravel irá determinar se uma versão compilada da visão existe. Se o arquivo existir, o Laravel irá então determinar se a visão não compilada foi modificada mais recentemente do que a visão compilada. Se a visão compilada não existe, ou se a visão não compilada foi modificada, o Laravel irá recompilar a visão.

Compilar visualizações durante a solicitação pode ter um pequeno impacto negativo no desempenho, então o Laravel fornece o comando Artisan view:cache para pré-compilar todas as visualizações utilizadas por seu aplicativo. Para aumentar o desempenho, você pode executar este comando como parte do seu processo de implantação:

- php artisan view:cache

Pode-se utilizar o comando view:clear para limpar o cache de visualizações:

- php artisan view:clear

Enviando dados para visualizações

Informações devem ser enviadas para uma view, através de uma matriz (array) de dados.

Ao passar informações dessa maneira, os dados devem ser uma matriz com pares de chave / valor. Depois de fornecer dados a uma visualização, você pode acessar cada valor em sua visualização usando as chaves dos dados, como:

Blade

Blade é o mecanismo de modelagem simples, mas poderoso, incluído no Laravel. Ao contrário de alguns mecanismos de modelagem de PHP, o Blade não o restringe de usar código PHP simples em seus modelos. Na verdade, todos os modelos Blade são compilados em código PHP simples e armazenados em cache até serem modificados, o que significa que o Blade adiciona essencialmente zero sobrecarga ao seu aplicativo.

Você não está limitado a exibir o conteúdo das variáveis passadas para a visualização. Você também pode reproduzir os resultados de qualquer função PHP. Na verdade, você pode colocar qualquer código PHP que desejar dentro de uma instrução echo Blade:

```
1 <h3>Data atual: {{ date("d/m/Y", time()) }}</h3>
```

Incluindo subviews

A maioria dos aplicativos da web mantém o mesmo layout geral em várias páginas. Seria incrivelmente complicado e difícil manter nosso aplicativo se tivéssemos que repetir todo o layout HTML em

cada visualização que criamos.

A diretiva @include do Blade permite incluir uma visualização do Blade dentro de outra visualização. Todas as variáveis disponíveis para a visualização pai serão disponibilizadas para a visualização incluída:

Mesmo que a visualização incluída herde todos os dados disponíveis na visualização pai, você também pode passar uma matriz de dados adicionais que devem ser disponibilizados para a visualização incluída.

Formulários - Campo CSRF

Sempre que definir um formulário HTML em seu aplicativo, você deve incluir um campo de token CSRF oculto no formulário para que o middleware de proteção CSRF possa validar a solicitação. Você pode usar a diretiva @csrf Blade para gerar o campo de token:

```
<form method="POST" action="/profile">
    @csrf

    ...
</form>
```

Diretivas Blade

Além da herança do template e da exibição de dados, o Blade também fornece atalhos convenientes para estruturas de controle PHP comuns, como instruções condicionais e loops. Esses atalhos fornecem uma maneira muito limpa e concisa de trabalhar com estruturas de controle do PHP;

Declarações if

```
@if (count($dados) === 1)
    Somente um registro!
@elseif (count($dados) > 1)
    Vários registros!
@else
    Nenhum registro!
@endif
```

De forma similar podemos utilizar a diretiva @unless:

```
@unless (Auth::check())
    Você não está logado!
@endunless
```

Além das diretivas condicionais já discutidas, as diretivas @isset e @empty podem ser usadas como atalhos convenientes para suas respectivas funções PHP:

```
@isset($dados)
    // $dados está definido e não vazio
@endisset

@empty($dados)
    // $dados está vazio...
@endempty
```

Diretivas de autenticação

As diretivas @auth e @guest podem ser usadas para determinar rapidamente se o usuário atual é autenticado ou é um convidado:

```
@auth
    // Usuário autenticado
@endauth

@guest
    // Usuário não autenticado
@endguest
```

Diretivas de seção

Você pode determinar se uma seção de herança de modelo tem conteúdo usando a diretiva @hasSection:

```
@hasSection('body')
  @yield('body')
@endif
```

Estruturas de decisão

Instruções switch podem ser construídas usando as diretivas @switch, @case, @break, @default e @endswitch:

```
@switch($i)
    @case(1)
        First case...
        @break

    @case(2)
        Second case...
        @break

    @default
        Default case...
@endswitch
```

Laços de iteração

Além de instruções condicionais, o Blade fornece diretivas simples para trabalhar com estruturas de loop do PHP. Novamente, cada uma dessas diretivas funciona de forma idêntica às suas contrapartes PHP:

```
@for ($i = 0; $i < 10; $i++)
    O valor atual é {{ $i }}
@endfor

@foreach ($dados as $item)
    <p>Nome: {{ $item->nome }}</p>
@endforeach

@forelse ($dados as $item)
    <li>{{ $item->nome }}</li>
@empty
    <p>Não há dados cadastrados</p>
@endforelse
```

```
14
15 @while (true)
16     <p>Eu sou um loop infinito </p>
17 @endwhile
```

Comentários

O Blade também permite definir comentários em suas visualizações. No entanto, ao contrário dos comentários HTML, os comentários do Blade não são incluídos no HTML retornado pelo seu aplicativo:

```
1 {{-- Isto é um comentário e não estará presente no HTML renderizado --}}
```

1.5 Sistema de Rotas

As rotas podem ser consideradas o espinha dorsal do backend, já que cada requisição que o servidor recebe é redirecionada para um controlador através de uma lista de roteamento que mapeia as requisições para os controladores ou ações.

Para criarmos as rotas para cada um dos métodos do controller, utilizaremos o arquivo routes/web.php

Este arquivo é responsável por rotear o caminho da request para a função correta no controller e retornar o resultado.

Como exemplo, supondo que tenhamos um controller chamado ControladorModalidades, as rotas para cada um dos métodos, seriam definidas como descrito abaixo:

```
1 Route::get('/modalidades', [App\Http\Controllers\ControladorModalidade::
    class, 'index'])->name('inicio');
2 Route::get('/modalidades/novo', [App\Http\Controllers\ControladorModalidade
    ::class, 'create'])->name('novo');
3 Route::post('/modalidades', [App\Http\Controllers\ControladorModalidade::
    class, 'store'])->name('salvar');
4 Route::get('/modalidades/apagar/{id}', [App\Http\Controllers\
    ControladorModalidade::class, 'destroy'])->name('apagar');
5 Route::get('/modalidades/editar/{id}', [App\Http\Controllers\
    ControladorModalidade::class, 'edit'])->name('editar');
6 Route::post('/modalidades/{id}', [App\Http\Controllers\ControladorModalidade
    ::class, 'update'])->name('atualizar');
```

Quando necessitarmos criar uma rota que não "passe" por nenhum controller, e retorne diretamente uma view, deveremos fazer isso, da seguinte forma:

```
1 Route::get('/', function () {
2     return view('index');
3 });
```

No exemplo acima, temos uma rota que retorna a view index, sem passar por nenhum controlador.