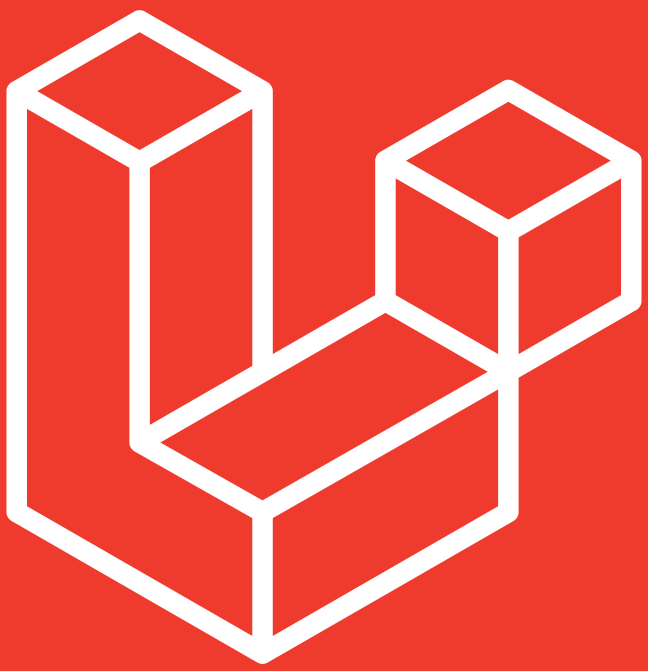


Laravel

Material de apoio ao desenvolvedor Laravel

MICILINI.COM

Introdução ao Laravel

O que é o Laravel?

É framework que facilita o desenvolvimento de aplicações em PHP, de modo a economizar tempo e custos para os desenvolvedores.

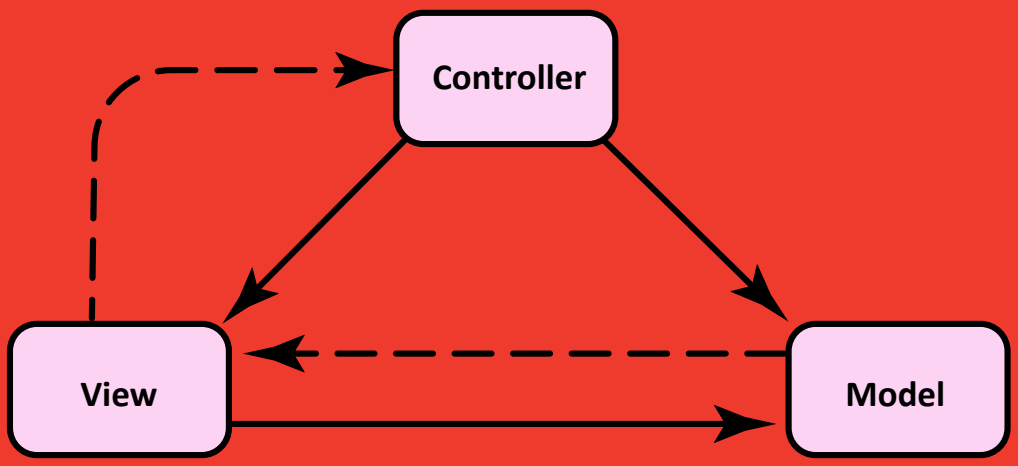
Ele conta com um conjunto de bibliotecas de modo a criar uma base aonde as aplicações são construídas. (Ou seja, muitas coisas já vem prontas).

Criar uma aplicação sem o Laravel está para cozinhar um bolo do zero, onde você precisa fabricar todos os ingredientes.

Já criar uma aplicação com o Laravel (ou qualquer outro framework) está para cozinhar um bolo com todos os ingredientes necessários em cima da mesa, bastando apenas cozinhar.

Estrutura do Laravel

O Laravel utiliza o padrão MVC (Model, View e Controller) que, basicamente, funciona da seguinte forma:



- Model é a camada responsável pela parte lógica da aplicação, ou seja, todos os recursos da sua aplicação (consultas ao BD, validações, notificações, etc.).

- View é a camada responsável por exibir dados para o usuário, seja em páginas HTML, JSON, XML, etc.

- Controller é o famoso “meio-de-campo” da aplicação. Essa é a camada que sabe quem chamar e quando chamar para executar determinada ação.

Configurações Iniciais

Configuração Inicial

É importante ressaltar que o Laravel contém um Build Server embutido, isso significa que você não precisa instalar a pilha LAMP (LINUX + PHP + APACHE + MYSQL), ou qualquer outro servidor local como é o caso do Xampp, Wamp e afins.

Mas sim, o projeto em Laravel consegue ser executado dentro da pilha LAMP ou dentro daquelas pastas (htdocs/www/public_html) existentes em servidores locais, compartilhados, e de produção.

O Laravel costuma funcionar melhor em servidores com NGINX do que servidores com Apache.

No caso do Apache, você terá que fazer certas modificações nos arquivos internos do Laravel para a execução ser possível.

Instalando o Laravel

Primeiro certifique-se de que o COMPOSER está instalado na sua máquina.

Em seguida abra o terminal (Prompt de Comando/CMD caso for windows) como administrador e instale globalmente o Laravel, com o seguinte comando:

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

Criando o primeiro projeto

Com o terminal aberto vá para um local onde você queira hospedar a sua primeira aplicação em laravel.

No meu caso eu escolhi o disco C:

```
cd C:/
```

Em seguida execute o comando laravel new, e informe o nome da pasta do seu projeto que você deseja criar (criei uma pasta chamada “projeto-laravel”):

```
laravel new projeto-laravel
```

Após essa grande instalação, vemos que ele instala todas as dependências que o laravel necessita para dar início a execução inicial do projeto.

Inicializando o Servidor

Considerando que estamos com o CMD ainda aberto dentro da pasta C:/.

Após a instalação do Laravel, foi criado uma pasta chamada “projeto-laravel”, sendo assim, agora precisamos entrar nela para podermos inicializar o servidor local (lembra que ele vinha com um Build Server Embutido?)

```
cd ./projeto-laravel
```

Em seguida utilizamos o ARTISAN para inicializar o projeto que acabamos de criar:

```
php artisan serve
```

Observação: Caso tenha dado algum erro alegando que o comando php não foi encontrado. Isso quer dizer que o php não está instalado na sua máquina local.

Em casos como esses (caso você estiver usando o Windows), você vai precisar instalar o Wamp ou Xampp (Recomendo o Xampp).

Após a instalação, basta fechar e abrir o Terminal (CMD), ir para o disco C, entrar na pasta do seu projeto, e executar novamente o comando php artisan serve.

Caso estiver com dificuldades, acesse este link.

A partir de agora uma mensagem será gerada alegando que podemos acessar o nosso servidor local em http://127.0.0.1:8000/.

Para desligar o servidor: feche o terminal ou use as teclas [CTRL] + [C].

O que é o Artisan?

Artisan é o nome da interface da linha de comando incluída no Laravel. Esta interface fornece um bom número de comandos auxiliares para que você use durante o desenvolvimento de sua aplicação.

Ou seja, como não estamos utilizando a pilha LAMP ou um Wamp/Xampp, o artisan fica responsável por chamar módulos do Laravel que inicializam o servidor.

Além disso, o artisan nos ajuda a criar controllers, models e afins de forma automática dentro das pastas do nosso projeto (Isso evita que criemos de forma manual).

Obviamente que em servidores de produção que utilizam o apache/nginx, não precisamos inicializar o servidor por meio do Artisan, pois o apache/nginx já reconheceria a executaria o projeto sozinho. (Ou seja, não precisaríamos executar o comando php artisan serve)

Mas como estamos trabalhando em um servidor de desenvolvimento, vale muito a pena utilizá-lo.

Visualizando a lista de comandos do Artisan

Para visualizar a lista disponível de comando do artisan você deve abrir seu CMD (não precisa estar na pasta do projeto, isto é, se você instalou o Laravel globalmente), e em seguida executar o comando:

```
php artisan
```

Você também pode executar os comandos - - help ou help para visualizar uma pequena ajuda com cada comando.

O exemplo abaixo foi utilizado esses dois comandos com o make:controller:

```
php artisan make:controller --help
php artisan help make:controller
```

Estrutura Interna do Laravel

Entendendo os diretórios

Se você abriu os arquivos existentes na pasta do Laravel, então chegou a hora de entendermos cada um deles:

README.md: arquivo markdown, muito utilizado no GitHub para adicionar a descrição do projeto [Só usamos ele para organizar informações para o GitHub].

App: a pasta mais **IMPORTANTE** do projeto, onde está as pastas do MVC e seus respectivos arquivos.

Artisan: pasta onde está localizada os arquivos da linha de comando do artisan (CLI).

Bootstrap: contém os arquivos de inicialização do projeto.

Composer.json e Composer.lock: arquivos do composer que armazenam as dependências que serão instaladas dentro do projeto.

Config: contém os arquivos de configuração do próprio Laravel, nas quais retornam arrays de configurações globais do seu projeto.

Database: contém arquivos especiais capazes de criar tabelas no banco de dados, inserir dados e afins, ou seja, manipulação do banco de dados no quesito estrutural.

Package.json: lista e mantém as dependências do projeto que foram feitas vida NodeJS com NPM.

Phpunit.xml: teste unitários que podem ser feitos usando o laravel.

Resources: local aonde ficam as Views do seu projeto (Neste Framework as views não ficam dentro e App, mas sim na pasta resources), com arquivos css, javascript e traduções.

Public: local aonde armazenamos arquivos públicos, como imagens, arquivos de download (PDF's, executáveis, etc...), Robots.txt, site_map.xml, favicon.ico e entre outros.

Routes: pasta aonde se localiza as rotas (Urls) do nosso projeto, onde estão categorizadas por rotas da web, rotas de api e afins.

Server.php: arquivo que ajuda o Build Server do PHP a emular o mod_rewrite do apache sem você precisar ter de fato um apache.

Storage: pasta aonde contém arquivos de sessão, logs, arquivos trazidos por meio de upload.

Tests: tem a ver com testes feitos dentro do projeto.

Vendor: pasta ande estão localizadas as dependências de terceiros que são gerenciadas pelo composer.

Observação 1: É importante ressaltar que algumas partes do front-end do Laravel estão inteiramente conectadas ao NodeJS por meio do arquivo mix chamado de webpack.mix.js.

Observação 2: No Laravel por padrão nos temos um template de front-end chamado Blade, isso significa que todas as views devem conter no final a seguinte extensão: ".blade.php".

Entendendo a Estrutura da pasta APP

Ela é uma pasta onde iremos concentrar 80% dos esforços do desenvolvimento de qualquer aplicação feita em laravel, a estrutura da pasta é esta:

Console: contém o arquivo kernel.php, aonde nos iremos poder criar nossos próprios comandos voltados ao artisan.

Exceptions: aqui podemos configurar a forma de captura das exceções que acontecem no projeto e o que fazer com elas.

Http: contem tudo o que tem a ver com a sua requisição e como você manipula tal requisição, os controllers e middlewares ficam aqui dentro.

Models: pasta ligada diretamente ao banco de dados, como comandos que fazem insert, select, update, delete e afins.

Providers: são provedores do laravel, aonde podemos configurar classes ou arquivos de apoio. Dependendo dos pacotes instalados o número de providers podem aumentar.

Colocando a Mão na Massa!

Introdução

A partir de agora, daremos início ao desenvolvimento com Laravel, começando pelas rotas, seguindo pelos controllers, Models, Providers... até você se sentir seguro para utilizar este framework de forma rápida.

Rotas com Laravel

Rotas (Web)

A gente começa o nosso desenvolvimento com Laravel pelas ROTAS, ou seja, primeiro a gente define uma rota e depois a gente se preocupa em criar os controllers, Models, Providers e afins.

As rotas no laravel elas ficam localizadas dentro da pasta routes.

Como vamos criar rotas para Web (não para API's), vamos trabalhar em cima do arquivo web.php existente dentro da pasta routes.

Se você abrir este arquivo (web.php), você verá uma rota simples:

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

Na instalação inicial, ele sempre cria uma rota do tipo GET, voltada a home ('/') aonde executa um call-back que retorna uma view chamada "welcome".

Esse arquivo de View ele está localizado dentro da pasta: *Resources > Views > Welcome.blade.php*.

Nesse caso, quando abrirmos a url http://127.0.0.1:8000, aquela rota será chamada, e o Laravel irá retornar ao usuário o conteúdo existente no arquivo welcome.blade.php

Criando uma rota simples

Para criar uma rota simples do tipo get, de forma que apareça uma mensagem de ‘Olá Mundo’ nos podemos usar a mesma estrutura do código anterior:

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

Tudo o que fizemos acima, foi criar primeiro uma nova rota do tipo GET, chamada de ‘/ola-mundo’.

Nesse caso, quando o usuário acessar a URL http://127.0.0.1:8000/ola-mundo, o laravel entre nessa rota e mostre a mensagem ‘Olá Mundo!’ ao usuário.

Criando uma rota e chamando uma View

Para criar uma rota de modo a carregar um “arquivo html”, sim, entre aspas porque no caso do Laravel o arquivo é PHP (mas que contém HTML dentro).

O primeiro passo que você precisa fazer é criar uma rota e usar o comando view(), para chamar um arquivo que deve existir dentro da pasta: *Resources > Views.*

```
Route::get('/abre-view', function (){
      return view('tela');
});
```

Importante: antes de executar o comando anterior, não se esqueça de que foi necessário criar um arquivo chamado **tela.blade.php** dentro da pasta *Resources > Views*.

Posso colocar o nome do arquivo completo dentro de View()?

Não, você não precisa informar o nome do arquivo completo (tela.blade.php) dentro do comando View(), pois o mesmo já possui um sistema interno de auto completar.

Neste caso, você só precisa informar tudo aquilo que vem antes do .blade.php como, por exemplo:

- Supondo que o nome do seu arquivo é **tela.blade.php**, você só precisa informar **view('tela')**.
- Supondo que o nome do seu arquivo é **meu_arquivo.blade.php**, você só precisa informar **view('meu_arquivo')**.
- Supondo que o nome do seu arquivo é **home-neutron.blade.php**, você só precisa informar **view('home-neutron')**.
- Agora supondo que seu arquivo esteja dentro de uma pasta chamada **home**, que existe dentro de **Views**, e seu arquivo se chama **inicio.blade.php**, você só precisa informar o caminho/nome: **view('home/inicio')**.

Verbose de Rotas (POST, PUT, DELETE...)

Além disso, nós podemos trocar o parâmetro get por alguns outros como, por exemplo:

- get
- post
- put
- patch
- delete
- options

Exemplo de uma rota com POST:

```
Route::post('/receber-nome', function (){
      return 'Seu nome é: '.$_POST['nome'];
});
```

Lembre-se que a rota acima não poderá ser acessada via URL (http://127.0.0.1:8000/receber-nome), pois se trata de uma rota do tipo POST, ou seja, usada somente para receber informações por meio de requisições (como formulários, por exemplo).

Trabalhando com Parâmetros Dinâmicos nas Rotas

Sabe aquelas rotas que podemos informar um parâmetro? Ou seja, um nome, uma categoria ou quem sabe até mesmo um número? Como, por exemplo:

➤ http://127.0.0.1:8000/produto/geladeira-multiuso

- http://127.0.0.1:8000/meu-nome-e/micilini

- http://127.0.0.1:8000/codigo/12775

Esses parâmetros nos auxilia no momento em que precisamos criar rotas que recebem parâmetros, como números, textos, caracteres, ID's e afins.

O parâmetro dinâmico é envolvido com chaves {} e dentro delas você deve informar o nome de variável que você quer receber.

Por exemplo:

```
Route::get('/meu-nome-e/{nome}', function ($nome){
      return 'Seu nome é: '.$nome;
});
```

Observe no código acima, que nós criamos uma rota que receberá um `{nome}`.

Observe também que criamos uma variável dentro de function chamada `$nome`, isso porque, tudo o que o usuário informar na URL ficará armazenado dentro dessa variável.

Observação: o nome da variável não precisa ser o mesmo do parâmetro dinâmico, mas fazemos isso por questões de padronização:

```php
Route::get('/idade/{numero}', function ($nao_precisa_se_chamar_numero){
  return 'Sua Idade é: '.$nao_precisa_se_chamar_numero;
});
```

O exemplo acima funciona, mas não segue a regra de padronização de nomes (você deve evitar isso).

Agora basta acessar as URL's para testar o que acabamos de criar:

- http://127.0.0.1:8000/meu-nome-e/micilini
- http://127.0.0.1:8000/idade/25

Parâmetros Dinâmicos Vazios

Pode acontecer do usuário às vezes não informar o parâmetro dinâmico, e tentar acessar a url sem informa-lo, como, por exemplo: http://127.0.0.1:8000/meu-nome-e/

Quando isso acontecer, o laravel vai dar um erro 404, uma vez que da forma como fizemos anteriormente, ele precisa receber esse parâmetro (É OBRIGATÓRIO!).

Agora, caso você queria fazer com que tal parâmetro seja opcional, você pode adicionar um [?] ao lado do parâmetro ({nome?}), e associar um valor padrão/nulo ao lado da variável $nome:

```php
Route::get('/meu-nome-e/{nome?}', function ($nome = null){
  return 'Seu nome é: '.$nome;
});//podemos informar $nome = 'Sem Nome' tambem...
```

Outros tipos de Parâmetros Dinâmicos

Rotas com expressões regulares? Rotas com múltiplos parâmetros? Isso é possível? Sim, vejamos abaixo como isso é feito:

```
Route::get('/user/{name}', function ($name){
  return $name;
})->where('name', '[A-Za-z]+');//Expressão regular em que o parâmetro name deve ser exclusivamente composto por letras


Route::get('/user/{id}', function ($id){
  return $id;
})->where('id', '[0-9]+');//Expressão regular em que o parâmetro id deve ser exclusivamente composto por números


Route::get('/user/{id}/{name}', function ($id, $name){
  return $id.'-'.$name;
})->where(['id' => '[0-9]+', 'name' => '[A-Za-z]+']);//Expressão regular em que estamos trabalhando com dois parâmetros, cada um com sua particularidade

Route::get('/user/{id}/{name}', function ($id, $name){
  return $id.'-'.$name;
});//Rota com dois parâmetros sem fazer o uso de expressões regulares
```

Delegando um Controller a uma Rota

A partir de agora nos iremos fazer com que uma rota carregue um determinado método específico de um **Controller**, ao invés de abrir uma view ou retornar um texto.

Ou seja, tudo o que nós iremos fazer agora é, assim que um determinado usuário abrir uma determinada URL no navegador, o Laravel chame um controller, mais especificamente um método existente dentro dele (padrão na maioria dos sistemas).

```php
Route::get('/ola-mundo', [\App\Http\Controllers\MeuController::class, 'meuMetodo']);//Criamos uma rota chamada ‘ola-mundo’, que chama o caminho aonde esta o meu Controller (’MeuController’), mais especificamente o método responsável (’meuMetodo’)

Route::get('/ola-mundo/{name}', [\App\Http\Controllers\MeuController::class, 'meuMetodoDois']);//Mesma lógica da rota anterior, a diferença é que estamos recebendo um parâmetro e chamando um novo método de um mesmo controller

Route::get('/ola-mundo/{id}', [\App\Http\Controllers\MeuController::class, 'meuMetodoTres'])->where('id', '[0-9]+');//Mesma lógica da rota anterior, a diferença é que estamos usando expressões regulares
```

Show! Agora que já sabemos como chamar um controller (apesar de não sabermos como criar um ainda), chegou o momento de aprender a criar um Controller :D

Controllers no Laravel

Local do Controller

Como dito anteriormente, os controllers da nossa aplicação estarão dentro da pasta: **app> http > Controllers**. Aonde lá existe um controller padrão da aplicação chamado de Controller.php (Não apague este arquivo, pois ele é necessário).

Criando um Controller

Existem duas maneiras de criar um controller, a primeira delas é fazendo isso na mão, criando um arquivo dentro daquela pasta Controller.

Como, por exemplo:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class MeuController extends Controller{

    public function meuMetodo(){
        return 'Olá Mundo!';
    }

}
```

No caso do comando acima, eu criei um novo arquivo chamado de **MeuController.php**, dentro da pasta *app > http > controllers*. Isso foi feito a mão para exemplificar que é possível criar um controller sem ser pelo Artisan.

A segunda maneira de se criar um controller, é fazendo isso diretamente pelo Artisan.

Primeiro se certifique de que você esta com o Terminal aberto na pasta do seu projeto.

Em segundo lugar, execute o comando:

```
php artisan make:controller MeuController
```

Automaticamente será um criado um controller na pasta **Controllers** cujo nome será: **MeuController**.

Não é necessário colocar a extensão **.php** no comando acima para criar um controller. Além disso, no local aonde está escrito **‘MeuController’** você poderá inserir o nome que desejar.

Carregando uma view pelo Controller

O processo de carregar uma view em um método existente em um controller, é bem simples, vejamos:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class MeuController extends Controller{

    public function meuMetodo(){
        return view('tela');
    }

}
```

Recebendo os Parâmetros da URL no Controller

Anteriormente nos vimos ser possível receber parâmetros dinâmicos nas nossas rotas, e aprendemos a como recuperar e usar eles. Só não vimos como fazer isso dentro dos controllers, certo? O processo é bem simples, vejamos:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class MeuController extends Controller{

    public function meuMetodo(Request $request){
        return $request->route('name');
    }

}
```

Tudo o que fizemos no comando anterior, foi passar na função a classe Request que é responsável por pegar todos os parâmetros de uma determinada requisição, e em seguida usar essa mesma classe para trazer o parâmetro **name** que veio pela URL.

Outra forma de recuperar o parâmetro sem ser pela classe Request é passando uma variavel no método chamado pelo controller:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class MeuController extends Controller{

    public function meuMetodo(Request $request, $nome){
        return $nome;
    }

}
```

Recebendo Parâmetros do tipo POST

No exemplo anterior, vimos como receber os parâmetros do tipo GET de uma determinada URL, a partir de agora, veremos como fazer isso com parâmetros do tipo POST.

No caso desses parâmetros, iremos precisar da classe Request, vejamos como isso é feito:

```php
public function meuMetodo(Request $request){
    $data = $request->post();
    return var_dump($data);
}
```

Por meio da função **post()** vista acima, estamos recuperando TODOS os possíveis parâmetros que estamos recebendo via método POST.

E se eu quiser recuperar somente 1 parâmetros específico, cujo nome é *'idade'*, por exemplo?

```php
public function meuMetodo(Request $request){
    $data = $request->post('idade');
    return $data;
}
```

Recebendo Parâmetros do tipo GET

Parâmetros do tipo GET são aqueles que vem na URL. Já vimos como recuperar no tópico 'Recebendo os parâmetros da URL no Controller'.

Mas existe uma outra forma de recuperar-los, vejamos:

```php
public function meuMetodo(Request $request){
    $data = $request->query();//Recebendo todos os parâmetros GET
    return var_dump($data);
}
```

```php
public function meuMetodo(Request $request){
   $data = $request->query('idade');//Recebendo um parâmetro especifico
   return $data;
}
```

Recebendo todos os tipos de parâmetros de uma só vez

Existe um comando chamado **all()**, que é responsável por recuperar todos os parâmetros que estão vindo em uma URL, sejam eles do tipo **GET, POST** e afins.

Veja como é fácil a utilização da função **all()**:

```
public function meuMetodo(Request $request){
    $data = $request->all();//Recebendo todos os parâmetros GET/POST
    return var_dump($data);
}
```

Models no Laravel

Local do Model

Como vimos anteriormente logo no início desse conteúdo, os Models são as classes aonde tratamos todo e qualquer tipo de comunicação com a nossa base de dados.

Os Models, eles ficam localizados dentro da pasta *App > Models*.

Criando um Model

Assim como nos controllers, existem duas maneiras de se criar um Model, uma delas é utilizando o Artisan:

```
php artisan make:model MeuModel
```

E a outra como já sabemos, é fazendo isso a mão. No exemplo abaixo eu criei dentro da pasta *Models* um arquivo chamado **MeuModel.php**:

```php
<?php

namespace App\Models;

use Illuminate\Http\Request;

class MeuModel extends Models{

}
```

OBSERVAÇÃO: Apesar de eu ter criado um Model chamado de 'MeuModel', existem algumas coisas que você precisa entender!

A primeira delas é que **o nome do model por padrão deverá ser o mesmo nome da sua tabela no banco de dados**.

O Laravel ele tem uma convenção padrão para cada Model. Por exemplo, quando criamos um Model chamado **User**, o laravel automaticamente entende que queremos trabalhar com a tabela **Users** (no plural) existente no banco de dados.

Quando criamos um model chamado **usuario**, o laravel entende que iremos trabalhar com a tabela **usuarios** que existe em nosso banco de dados.

É possível trocar o nome da tabela que será usada no Model?

Supondo que você queria deixar o nome do seu Model como **'Meu Model'**, enquanto gostaria de usar a tabela **usuários**... isso é possível? **SIM!**

Para fazer isso, basta informar uma variável de escopo chamada **$table**, aonde dentro dela você deverá informar o nome da sua tabela:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class MeuModel extends Models{
    use HasFactory;
    protected $table = 'nome-da-minha-tabela';

}
```

Por de baixo dos panos, o Laravel já conta com essa variável chamada de **$table**, então no caso, só estamos reescrevendo ela.

Importante: Desativando o TimeStamps!

O laravel conta com uma automatização de modo que sempre que você fizer um INSERT ou um UPDATE, no final da sua query ele adicione/atualize mais dois campos automaticamente para você, que são:

- created_at
- updated_at

Só que nem sempre temos esses dois comandos existem dentro de cada uma de nossas tabelas, ou talvez, o nome deles estejam em um outra tradução como ‘criado_em’ | ‘atualizado_em’.

Nesses casos, para desativarmos essa automatização usamos a variável de escopo: `public $timestamps = false;`

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class MeuModel extends Models{
    use HasFactory;
    public $timestamps = false;

}
```

Com este comando, estamos desativando o timestamps do Laravel. E precisamos fazer isso em todo Model que você criar.

Trabalhando com Models

Com o Laravel não precisamos ficar criando Querys e métodos dentro dos nossos Models, uma vez que ele já conta com todas essas automatizações por de baixo dos panos.

Fazendo com que só seja necessário chamarmos o Model junto a algumas funções já disponibilizadas pelo Laravel.

Considerando que temos uma tabela no banco de dados chamada de **usuarios**, com as seguintes colunas:

☐	1	ID_Usuario 🔑	int(11)
☐	2	NM_Nome	varchar(255)
☐	3	NR_Idade	int(11)
☐	4	DT_Cadastro	datetime

E que também temos uma nova rota chamada de **banco-de-dados**, que aponta para um controller chamado de **MeuController**:

```
Route::get('/banco-de-dados',
[\App\Http\Controllers\MeuController::class, 'meuBanco']);
```

Vamos supor também que já temos um Model chamado de usuario, com as seguintes configurações:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class usuario extends Models{
    use HasFactory;
    public $timestamps = false;

}
```

Para usar este Model, você só precisa importar dentro do **MeuController**, e em seguida fazer uma referência a ele dentro de um dos seus métodos (no meu caso estou usando o método meuBanco):

```
use App\Models\usuario;
```

```
public function meuMetodo(){
    $usuarioModel = new usuario();
    $dados = $usuarioModel::all();//com o all, puxamos todos os dados da
    tabela 'usuarios'
    return var_dump($dados);
}
```

Existe também outra forma de chamar o Model usuario, sem a necessidade de usar o comando `use App\Models\usuario;`

Para isso, basta apenas que você chame diretamente no pelo método dessa forma:

```php
public function meuMetodo(){
   $dados = App\Models\usuario::all();
   return var_dump($dados);
}
```

Veja como ficou o arquivo final (MeuController):

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Models\usuario;

class MeuController extends Controller{

    public function meuMetodo(){
        $usuarioModel = new usuario();
        $dados = $usuarioModel::all();
        return var_dump($dados);
    }

}
```

Agora caso você queria somente que retorne alguns dados da tabela, você pode passar isso entro do método **all()**, informando o array das colunas que você deseja retornar:

```php
public function meuMetodo(){
   $dados = App\Models\usuario::all(['NM_Nome', 'DT_Cadastro']);
   return var_dump($dados);
}
```

Além disso, podemos fazer um select de forma diferenciada usando a cláusula WHERE, aonde iremos pegar os registros do usuário cujo **ID_Usuario** for igual a 1:

```
public function meuMetodo(){
    $dados = App\Models\usuario::where('ID_Usuario', 1)->get();
    return var_dump($dados);
}
```

Para mais dicas sobre como usar as classes dos Models no Laravel acesse este link!

Como Fazer Insert?

No caso do **INSERT**, ele acontece de maneira bem fácil, observe:

```
public function meuMetodo(){
    $usuario = new Usuario;
    $usuario->NM_Usuario = 'Micilini';
    $usuario->NR_Idade = 26;
    $usuario->DT_Cadastro = 'timestamp-data-de-hoje';
    $usuario->save();

    return 'Novo Usuário Salvo!';
}
```

Se quisermos que retorne o último id que foi inserido basta que após o **save()** usemos o comando:

```
public function meuMetodo(){
    $usuario = new Usuario;
    $usuario->NM_Usuario = 'Micilini';
    $usuario->NR_Idade = 26;
    $usuario->DT_Cadastro = 'timestamp-data-de-hoje';
    $usuario->save();

    $idInserido = $usuario->id;//Retorna o ID_Usuario inserido

    return 'Novo Usuário Salvo! ID = '.$idInserido;
}
```

Como Fazer Update?

No caso do **UPDATE**, ele acontece de maneira bem fácil, observe:

```
public function meuMetodo(){
    $usuario = new Usuario;
    $usuario::where('ID_Usuario', '=', 1)->update(array(
    'NM_Nome' => 'Micilini Roll',
    'NM_Idade' => 27
    ));

    return 'O Usuário de ID (1) foi Atualizado!';
}
```

Outra maneira de fazer atualizações é usando o comando **find()** seguido do **save()**, vejamos:

```
public function meuMetodo(){
    $usuario = new Usuario;
    $usuario = Usuario::find(1);//Encontra o ID_Usuario 1
    $usuario->NR_Idade = 28;
    $usuario->save();

    return 'O Usuário de ID (1) foi Atualizado!';
}
```

Como Fazer Delete?

No caso do **DELETE**, ele acontece de maneira bem fácil, observe:

```
public function meuMetodo(){
    $usuario = new Usuario;
    $usuario::where('ID_Usuario', '=', 1)->delete();

    return 'O Usuário de ID (1) foi Deletado!';
}
```

Também podemos fazer isso usando o **find()**, observe:

```php
public function meuMetodo(){
    $usuario = new Usuario;
    $usuario = Usuario::find(1);//Encontra o ID_Usuario 1
    $usuario->delete();

    return 'O Usuário de ID (1) foi Deletado';
}
```

Query Builder

O que é?

Anteriormente nós vimos o quão fácil é trabalhar com inserções, seleções, atualizações e remoções no Laravel, basta apenas que criemos um Model com o mesmo nome da tabela que queremos usar e pronto, o laravel já te dá todos os métodos necessários.

Só que existe um porém... e se eu quiser escrever querys a moda antiga?, ou seja:

```
‘SELECT * FROM usuarios WHERE ID_Usuario = 1’
```

Isso é possível? Sim! E tudo isso graças ao Query Builder!

O Query Builder fornece uma interface conveniente e fluente para criar e executar consultas de banco de dados.

Ele usa a ligação ao parâmetro PDO para proteger sua aplicação contra-ataques de injeção de SQL. Removendo a necessidade de limpar ou higienizar as variáveis passadas para o construtor de consultas.

Iniciando com Query Builder

No caso do Query Builder, não temos mais a necessidade de criar um Model do mesmo nome da tabela que queremos usar no banco de dados.

A sua utilização é bem simples, e o primeiro passo é inserir a classe **DB()** no seu Controller (ou no arquivo onde você deseja fazer consultas no banco de dados):

```
use Illuminate\Support\Facades\DB;//Importa o Query Builder
```

Veja como ficou a importação da classe no **MeuController**:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;//Importa o Query Builder

class MeuController extends Controller{

    public function meuMetodo(){
        return 'Olá Mundo!!';
    }

}
```

Fazendo um SELECT (Query Builder)

Para executarmos um SELECT com o Query Builder, é bem fácil, vejamos:

```
DB::select("SELECT * FROM usuarios WHERE ID_USUARIO = :idUsuario",
['idUsuario' => 1]);
```

No caso do comando acima, ele retorna um array do tipo **STDClass**, para convertê-lo em um tipo de array que seja fácil utilização podemos usar o comando **decode/encode** do **JSON**:

```
$data = json_decode(json_encode(DB::select("SELECT * FROM usuarios
WHERE ID_USUARIO = :idUsuario", ['idUsuario' => 1]));
var_dump($data);
```

Observe no select acima que nós estamos protegendo o nosso select usando parâmetro do tipo **bind (:idUsuario)**, o que nos protege contra SQL Injection.

Diferente do comando abaixo, que NÃO PROTEGE de SQL Injection:

```
DB::select("SELECT * FROM usuarios WHERE ID_USUARIO = 1");
DB::select("SELECT * FROM usuarios WHERE ID_USUARIO = $idUsuario");
```

As querys que acabamos de ver acima, são conhecidas no mundo do Laravel como Query Builder do tipo RAW.

Existe também a possibilidade de usar o Query Builder sem ser do tipo RAW, para isso podemos fazer da seguinte forma que é um pouco mais automatizada:

```
$todosOsDados = DB::table('usuarios')->where('ID_Usuario', '=',
1)->get();

$nomeUsuario = DB::table('usuarios')->where('ID_Usuario', '=',
1)->pluck('NM_Nome');
```

Tenha em mente que o retorno será também um array do tipo **STDClass.**

Fazendo um INSERT (Query Builder)

Para executarmos um INSERT (RAW) com o Query Builder, é bem fácil, vejamos:

```
DB::insert("INSERT INTO usuarios (NM_Nome, NR_Idade, DT_Cadastro)
VALUES (?, ?, ?)", ['Fulano', 22, 'timestamp-da-date']);
```

E se eu quiser retornar o ID que foi inserido? Basta usar ao final dessa query o seguinte comando:

```
$id = DB::getPdo()->lastInsertId();
```

Para executar um INSERT (sem RAW) com o Query Builder:

```
DB::table('usuarios')->insert([
    'NM_Nome' => 'Fulano',
    'NR_Idade' => 43,
    'DT_Cadastro' => 'timestamp-da-date'
]);//Troque 'insert' por 'insertGetId' para retornar o id inserido!
```

Observação: Tanto as querys com ou sem RAW (vistas acima) estão protegidas contra SQL Injection.

Fazendo um UPDATE (Query Builder)

Para executarmos um UPDATE (RAW) com o Query Builder, é bem fácil, vejamos:

```
DB::update("UPDATE usuarios SET NM_Nome = ? WHERE ID_Usuario = ?",
['Fulaninho de Tal', 1]);
```

```
DB::update("UPDATE usuarios SET NM_Nome = :nome WHERE ID_Usuario =
:id", ['nome' => 'Funkyo', 'id' => 1]);
```

Para executarmos um UPDATE (sem RAW) com o Query Builder

```
DB::table('usuarios')->where('ID_Usuario', 1)->update([
    'NM_Nome' => 'Fulano Xyz',
    'NR_Idade' => 33
]);
```

Observação: No comando acima estamos utilizando o **where** sem qualquer tipo de operador relacional **(=, >, <, !=, <=, >=)**, isso significa que por padrão estamos usando o operador de igualdade (=).

Mas você também pode adicionar um operador ali se quiser, por exemplo:

```
DB::table('usuarios')->where(['ID_Usuario', '>=', 1])->update([
    'NM_Nome' => 'Fulano Xyz',
    'NR_Idade' => 33
]);
```

E se eu quiser adicionar mútiplos wheres? É fácil, veja como fazer abaixo:

```
DB::table('usuarios')->where(['ID_Usuario', '>=', 1], ['status', '=',
false])->update([
    'NM_Nome' => 'Fulano Xyz',
    'NR_Idade' => 33
]);
```

Observação: Tanto as querys com ou sem RAW (vistas acima) estão protegidas contra SQL Injection.

Fazendo um DELETE (Query Builder)

Para executarmos um DELETE (RAW) com o Query Builder, é bem fácil, vejamos:

```
DB::delete("DELETE FROM usuarios WHERE ID_Usuario = ?", [1]);
```

Para executarmos um DELETE (sem RAW) com o Query Builder, é bem fácil, vejamos:

```
DB::table('usuarios')->where(['ID_Usuario', '=', 1])->delete();
```

Observação: Tanto as querys com ou sem RAW (vistas acima) estão protegidas contra SQL Injection.

Criando um Model Único (DatabaseModel)

Até aqui você viu como executar querys e fazer chamadas no banco de dados por meio dos controllers.

Uma dica legal, é você criar um Model, onde você poderá criar diversos métodos responsáveis por fazer consultas na base de dados. Isso tiraria os comandos de consulta de banco de dentro do seu controller e jogaria para dentro de um Model específico.

De modo que a partir de agora, os métodos do seu controller, executariam os métodos de um determinado model, que por sua vez iria realizar consultas na sua base de dados.

Veja como é fácil:

```
php artisan make:model DatabaseModel
```

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class DatabaseModel extends Models{
    use HasFactory;
    public $timestamps = false;


    public function consultarUsuario(){
        $data = json_decode(json_encode(DB::select("SELECT * FROM
        usuarios WHERE ID_USUARIO = :idUsuario", ['idUsuario' => 1])));
        return $data;
    }

    public function inserirUsuario(){
        DB::table('usuarios')->insert([
            'NM_Nome' => 'Fulano',
            'NR_Idade' => 43,
            'DT_Cadastro' => 'timestamp-da-date'
        ]);
    }

    public function updateUsuario(){
        DB::table('usuarios')->where(['ID_Usuario', '>=', 1])->update([
            'NM_Nome' => 'Fulano',
            'NR_Idade' => 43
        ]);
    }

    public function consultarStatusAtivo(){
        $data = json_decode(json_encode(DB::select("SELECT * FROM
        status WHERE IS_Removido = true")));
        return $data;
    }

    public function deletarUsuario(){
        ....
    }

    public function deletarStatusUsuario(){
        ....
    }



}
```

Trabalhando com Views

O que são?

Como vimos anteriormente, as Views são arquivos PHP, que contém códigos HTML, Javascript e CSS, além de poder conter (é claro) código PHP também.

O Laravel utiliza o template Blade para a criação de Views, não é atoa que toda view termina com a extensão .blade.php.

Blade é o mecanismo de modelagem simples, mas poderoso, incluído no Laravel. Ao contrário de alguns mecanismos de modelagem PHP, o Blade não o restringe de usar código PHP simples em seus modelos.

Na verdade, todos os modelos do Blade são compilados em código PHP simples e armazenados em cache até serem modificados, o que significa que o Blade adiciona basicamente zero sobrecarga ao seu aplicativo.

Os arquivos de modelo blade usam a extensão de arquivo .blade.php e são geralmente são armazenados no diretório resources/views.

Criando uma View

O processo de criação de uma View é bem simples, e pode ser feito pelo Artisan com o seguinte comando:

```
php artisan make:view minhaView
```

Com o comando anterior estamos criando um novo arquivo chamado **minhaView.blade.php** dentro da pasta *resources > Views.*

Para criar uma view pelo artisan em um subdiretório da pasta views, basta adicionar o ponto (.) antes de informar o nome da view:

```
php artisan make:view pasta.minhaView
```

Para criar uma View com outro tipo de extensão, basta usar o comando: (aqui estamos criando com a extensão .html)

```
php artisan make:view outraView --extension=html
```

Usando uma View

Para usar uma view é bem simples, até porque já fizemos isso antes :D

Você pode abrir uma view tanto dentro de um método de um Controller quanto dentro de uma rota no arquivo web.php existente dentro da pasta Routes.

```php
return view('minhaView');

return view('minhaPasta/minhaView');//Caso sua view esteja dentro de uma pasta
```

Lembre-se que você não precisa colocar a extensão .blade.php dentro do comando view, uma vez que ele já faz isso por baixo dos panos.

Passando Parâmetros para a View

Supondo que você queira passar alguma informação para dentro da sua tela, você pode fazer isso dentro do comando **view()**, veja como é simples:

```php
return view('minhaView', ['nome' => 'micilini', 'idade' => 25]);
```

As informações são repassadas para dentro da sua tela em formato de array, e a chave (key) do array, se transformá em um nome de uma variável que poderá ser chamada dentro da sua tela, por meio de chaves duplas.

Por exemplo, acima nós estamos passando a chave (key) chamada ‘nome’ que armazena o valor ‘micilini’.

Dentro da sua tela, a **chave (key)** poderá ser acessada informando o **sifrão ($)** mais o **nome da chave (key)**. Ou seja, como se estivéssemos escrevendo o nome de uma variável em PHP.

A diferença é que chamamos ela dentro de chaves duplas, observe como foi feito no arquivo **minhaView.blade.php**:

```blade
<!DOCTYPE html>
<html>
<body>

<h1>Meu nome é {{ $nome }}, e a minha idade é {{ $idade }}</h1>

</body>
</html>
```

Trabalhando com Views

O template Blade possui algumas funcionalidades que nos ajudam no dia a dia da nossa aplicação, vejamos algumas.

If e Else:

```
@if($nome == 'micilini')
<p>Seja bem vindo Micilini</p>
@else
<p>Você não é o Micilini...</p>
@endif
```

Foreach:

```
@foreach($meuArray as $array)
<p>Olá: {{ $array }}</p>
@endforeach
```

While:

```
@while($idade < 18)
<p>Você não tem idade suficiente...</p>
@endwhile
```

For:

```
@for ($i = 1; $i <= (5 - $post->images->count()); $i++)
    <div class="col"> </div>
@endfor
```

Definindo uma ÚNICA variável, método 1:

```
@define $nomeDaVariavel = 'Valor da Variável';
```

Definindo uma ÚNICA variável, método 2:

```
@php ($nomeDaVariavel = 'Valor da Variável')
```

Definindo múltiplas variáveis:

```
@php
   $valorUm = 1;
   $valorDois = 2;
@endphp
```

Escrevendo códigos PHP dentro das Views

Sim, esse tipo de coisa é totalmente possível, o primeiro método seria abrindo e fechando as tags do PHP, como se você estivesse escrevendo um código PHP normalmente:

```
<!DOCTYPE html>
<html>
<body>

<?php
 //aqui dentro você pode criar variáveis, funções, classes e seguir a
 lógica que você quiser...
?>

<h1>Meu nome é {{ $nome }}, e a minha idade é {{ $idade }}</h1>

</body>
</html>
```

O segundo método de fazer isso é abrindo as tags PHP da mesma forma como fizemos no subtópico de **‘Definindo múltiplas variáveis’**:

```
@php
   //aqui dentro você também pode criar variáveis, funções, classes e
   seguir a lógica que você quiser...
@endphp
```

Definindo uma função dentro de uma View:

```
@php
if ( !function_exists( 'mytemplatefunction' ) ) {
   function mytemplatefunction( $param ) {
      return $param . " World";
   }
 }
@endphp

<p>{{ mytemplatefunction("Hello") }}</p>
```

Chamando um método de uma classe dentro de uma View:

Também é possível chamar um método existente dentro de uma Classe (Controller, Model, Provider...) dentro de uma view.

Para exemplificar, vamos supor que criamos um método chamado **somarMaisDois** dentro do nosso controller **MeuController**:

```
...

public static function somarMaisDois($numero){
        return $numero + 2;
}

...
```

O Objetivo desse método é receber um número, somar mais 2 e retornar o resultado dessa equação. Lembre-se que o método precisa ser do tipo ‘public’ (público).

Por fim, basta chamarmos dentro das chaves duplas o caminho completo do Controller, fazendo referência ao método, veja como é fácil:

```
<p>{{ App\Http\Controllers\MeuController::somaMaisDois(5) }}</p>
```

Importanto outras views dentro de uma view

No template Blade, podemos usar o comando **@include()**, para fazer a inclusão de uma view externa dentro da sua view atual:

```
@include('minhaSegundaView')
```

Observe que não é necessário informar a extensão do arquivo **(.blade.php).**

Lembre-se também que estamos chamando a view chamada **'minhaSegundaView'** que está dentro da pasta *views*.

Mas e se eu quiser adicionar uma view que existe em uma determinada pasta existente dentro da pasta views?

Para isso basta escrever o caminho separado por pontos (.):

```
@include('pasta.minhaTerceiraView')

@include('pasta.subpasta.minhaQuartaView')
```

Esse tipo de comando é bastante utilizado quando queremos adicionar um Header ou um Footer que nunca muda.

Configurações do Framework

Arquivo .ENV

Este é o arquivo principal das configurações que serão consumidas por cada parte do projeto do Laravel, ou seja, ele estará acessível em cada controller, cada model, cada view e em outros arquivos também.

Aqui podemos configurar a parte de email, tempo de vida da session, do cookie, da página inicial do app, conexões com o banco de dados e entre outras customizações personalizadas.

É importante ressaltar que o laravel ele não lê as configurações a partir deste arquivo (de forma direta), mas sim dos arquivos existentes na pasta Config.

Criando uma configuração global

O arquivo .env, está localizado na pasta raiz do seu projeto Laravel, e ele tem mais ou menos essa cara:

```
APP_NAME='Micilini'
APP_ENV=local
APP_DEBUG=true
APP_URL=http://127.0.0.1:8000

LOG_CHANNEL=stack
LOG_DEPRECATIONS_CHANNEL=null
LOG_LEVEL=DEBUG
....
```

Caso desajar, você pode, por exemplo, criar uma nova variável global no final do arquivo, de modo que você tenha acesso a ela, em seus Controllers, Views e Models.

Por exemplo, vamos criar uma nova variável chamada de 'NOME':

```
NOME=micilini
```

Você também pode se quiser envolver o valor com aspas simples:

```
NOME='micilini'
```

Para ter acesso a esta variavel, você pode chama-la por meio do comando global **env()**, tanto em arquivos PHP:

```
env('NOME');
```

Quanto dentro das views usando o template Blade:

```
{{ env('NOME') }}
```

Banco de dados

As configurações necessárias para se realizar a comunicação com o banco, existem dentro do arquivo .ENV mais especificamente nessas variáveis:

```
DB_CONNECTION=
DB_HOST=
DB_PORT=
DB_DATABASE=
DB_USERNAME=
DB_PASSWORD=
```

Desativando/Ativando os erros do Laravel

A configuração responsável por ativar/desativar os erros que podem acontecer na sua aplicação é o `APP_DEBUG` use `TRUE` para ativar os erros, e `FALSE` para desativá-los.

Mudando a URL da sua aplicação

A configuração responsável por alterar a URL da sua aplicação é o `APP_URL` que por padrão está setado para `http://127.0.0.1:8000`

Nós costumamos usar essa variável (**APP_URL**) dentro de nossas views para montarmos os links que levam os usuários para as outras páginas, por exemplo:

```
<!DOCTYPE html>
<html>
<body>

<a href="{{ env('APP_URL') }}/home">Home</a>
<a href="{{ env('APP_URL') }}/contato">Contato</a>

<h1>Meu nome é {{ $nome }}, e a minha idade é {{ $idade }}</h1>

</body>
</html>
```

Desse modo, se precisarmos trocar a URL para algum outro domínio, basta alterarmos o valor da variável **APP_URL** que automaticamente todas as páginas serão atualizadas.

Como mudar a porta da aplicação?

E se eu quiser abrir a minha aplicação na porta 7812, em vez da porta 8000? Isso é possível? Sim, e você pode fazer isso em duas etapas:

Primeiro altere a porta da variável APP_URL, de 8000 para a porta desejada:

```
APP_URL=http://127.0.0.1:7812
```

Em seguida termine o servidor pelo seu terminal (CTRL + C), e depois inicie o servidor novamente com flag **--port=PORTA**

```
php artisan serve --port=7812
```

Observação: Você sabia que é possível executar simultaneamente diversos projetos Laravel? Para isso basta seguir a lógica acima, aonde cada projeto é aberto em uma porta especifica :D

Migrations

O que são Migrations?

A migração do Laravel é uma forma que permite a você criar uma tabela em seu banco de dados, sem realmente ir ao gerenciador de banco de dados como phpmyadmin ou sql lite ou qualquer que seja o seu gerenciador.

Ou seja, nada mais são do que arquivos de banco e dados que são salvos dentro da pasta *Database > Migrations*.

Ali dentro você encontra diversos arquivos do tipo PHP que levam o nome das tabelas que deverão existir no seu banco de dados. Junto a isso, cada arquivo começa com uma data específica, indicando para o Laravel a ordem que tudo deverá ser criado.

Criando uma Migration

Supondo que o seu projeto precise de uma tabela chamada usuarios no banco de dados.

Com o terminal (CMD) aberto dentro da pasta do seu projeto, digite o seguinte comando:

```
php artisan make:migration create_usuarios_table
```

Tenha em mente que entre o **create_** e o **_table** existira o nome da sua tabela, essa nomenclatura já cria de automático um arquivo de migration, onde já vem configurado algumas informações como id e timestamp:

```php
public function up()
{
    Schema::create('usuarios', function (Blueprint $table){
        $table->id();
        $table->timestamps();
    });

}
```

Para continuar a setar as colunas dessa tabela é só seguir a lógica da variável **$table**, por exemplo:

```php
public function up()
{
    Schema::create('usuarios', function (Blueprint $table){
        $table->id('ID_Usuario');
        $table->string('NM_Nome');
        $table->integer('NR_Idade');
        $table->date('DT_Cadastro');
    });

}
```

Como executar uma Migration

Considerando que queiramos fazer a migração dos 3 arquivos padronizados que foram criados de forma automática pelo Laravel na pasta *Database > Migrations*.

Tudo que precisamos fazer é:

1-> Com o CMD aberto entre na pasta do nosso projeto
2-> Execute o seguinte comando do artisan:

```
php artisan migrate:install
```

Automaticamente o seu banco de dados local (Mysql, PhpMyAdmin e afins) acabou de receber uma nova tabela chamada Migrations.

Esse comando ele não instala todas as tabelas que estamos vendo na pasta **database > Migrations**, ok?

Tudo o que ele fez foi criar uma tabela de Migrations que funciona como uma espécie de Log, ou seja, para registrar/contabilizar o histórico das tabelas criadas pelo laravel.

Ele salvara isso dentro de um Id (identificador único), migration (nome do arquivo que foi executado), e o batch (lote em que isso foi executado, isto é, caso formos instalar diversas tabelas em um único comando).

Isso serve para que na próxima migração, ele não execute os arquivos que já foram executados, evitando assim que ocasione um erro no desenvolvimento.

Agora para instalar os arquivos existentes na pasta **Database > Migrations**, basta executar no CMD o comando:

```
php artisan migrate
```

Um ponto interessante é que este comando também substitui o que digitamos acima, ou seja, caso você execute ele antes de fazer **migrate:install**, o artisan é suficientemente inteligente para verificar se existe a tabela migrations (e se não existir ele cria), antes de importar os arquivos existentes em **Database > Migrations**.

Tinker

O que é?

O Laravel Tinker permite que você interaja com um banco de dados sem a necessidade de ter que escrever funções que façam esse tipo de coisa no seu código.

Sabe quando você quer fazer um insert, ou quem sabe um select só para ver como esses dados estão vindo e se comportando?

Então, em casos assim você simplesmente entra no seu código, cria um model, conecta com um controller, cria uma rota, escreve o código do select e depois acessa o localhost para ver como ele veio, certo?

E se eu te disser que por meio do artisan, vocês mesmo pode escrever esse código do select, de modo a verificar como ele está vindo, isso diretamente pelo terminal?

Sim isso é possível com o tinker, e isso significa que ele é uma mão na roda.

Obviamente que ele não adiciona códigos ou faz alterações no seu projeto, mas, se você executar um comando insert,update ou delete, aí, sim, talvez isso seja alterado na sua tabela do banco e dados... então CAUTELA.

Executando o Tinker

Para executar o Tinker, certifique-se de que você está com o CMD aberto, e que você esteja dentro da pasta raiz do seu projeto, após isso basta executar o comando:

```
php artisan tinker
```

Automaticamente será aberto um input aonde podemos preencher algumas informações, aqui podemos brincar de criar variáveis e até mesmo mostrar no terminal, veja como é fácil:

```
>>> $id = 1;
=> 1
>>> print $id;
1
=> 1
```

Obviamente que o Tinker não foi feito pra isso, então para iniciar a brincadeira, você pode, por exemplo, colar um código de select:

```
App\Models\Mundo::all();
```

Que o tinker se encarregará de mostrar e retornar ali mesmo no terminal os valores dessa requisição, viu como é fácil?

Acredito que a partir de agora você já saberá como trabalhar com Laravel da melhor forma, bons estudos :D

Gostou desse material?

Então não deixe de acessar a nossa seção de PHP.