

**1° Verificando a conexão com o banco de dados**

PHP ARTISAN MIGRATE:INSTALL

(dentro do banco irá criar um arquivo Migrations que listará as nossas migrações de Controle)

Successfully, está conectado a sua base de dados;

**2° Criando minhas Migrations:**

(files) na pasta Migrations:

Que serão as tabelas do banco de dados:

PHP ARTISAN MAKE: Create\_Table\_Stores

* Crie as estruturas da tabela da sua base
* Defina as chaves estrangeiras foreing

PHP ARTISAN MAKE: Create\_Products\_Table

* Crie as estruturas da tabela da sua base
* Defina as chaves estrangeiras foreing

Depois de criar as suas bases, rode para criar as tabelas dentro do Banco:

PHP ARTISAN MIGRATE

Check no banco de dados a criação das tabelas Stores, Products, será criado também outras tabelas como **failed\_jobs**, e **password\_resets.**

**3° Comandos Adicionais para Migrations:**

**Revertendo os dados:**

O comando Rollback volta um passo:

PHP ARTISAN MIGRATE:ROLLBACK

**Cont. Revertendo dados:**

O comando Rollback associado com –Steps=2 permite retroceder a quantidade de passos

PHP ARTISAN MIGRATE:ROLLBACK --STEPS=2

4° **Re-executando as Migrações:**

**Refazendo Dados:**

O comando fresh dará um Drop nas tabelas e recria elas sem adicionar valores.

PHP ARTISAN MIGRATE:FRESH

**5° Atualizando sem dar Drop nas tabelas:**

**Atualizando Dados:**

O comando refresh apenas atualiza a lista de tabelas sem adicionar valores a elas

**6° Atualizando inserindo valores:**

Podemos fazer da seguinte forma:

PHP ARTISAN MIGRATE:REFRESH --SEED (refaz as tabelas e adiciona dados no final)

PHP ARTISAN DB:SEED (Adiciona Valores as tabelas)

PHP ARTISAN MIGRATE: REFRESH STEPS=2

Esta ultima opção volta 2 passos.



# SEEDS:

**7° O que são e como utilizar:**

As Seed são os dados que vão alimentar as suas tabelas com o banco de Dados, elas permitem criar dados para nossas tabelas. Temos um arquivo principal dentro da pasta **SEEDS** o (**DatabaseSeeder.PHP**) este arquivo contém o Metodo **Run**(), que executa uma chamada de classe ou Namespace e a função **Call**:



**8° Para criarmos um arquivo Seed utilizamos o comando no terminal:**

PHP ARTISAN MAKE:SEEDER UserTableSeeders

**9° Para refazermos uma inserção de dados com seeds usamos o comando –Seed**

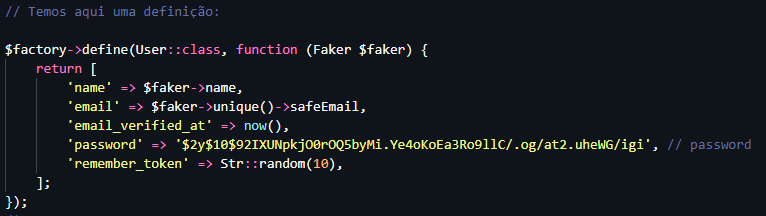
PHP ARTISAN MIGRATE:REFRESH –SEED

**MODEL FACTORYS**

**10° As factorys são auxiliares para nossas SEEDs:**

As factorys definem modelos para que possa fazer a criação de uma massa grande de dados

Dentro do arquivo UserFactorys na pasta Factories, temos exemplos disto:



utilizando a função factory(namespace da classe, quantidade de dados)->método create:

Podemos facilmente, criar a quantidade de registros que precisamos, as factorys se tornam muito uteis no quesito quantidade de dados para nosso banco.



PHP ARTISAN MIGRATE:REFRESH –SEED

Insere 40 usuários no banco de dados (stores)

PHP ARTISAN DB:SEED

Irá inserir mais 40 usuários em nossa tabela.

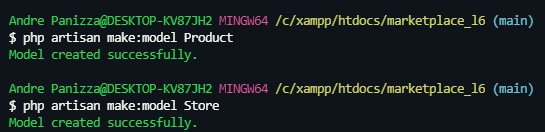
**GERANDO MODELS**

**11° Para geração desses Models, que vão ficar localizados na pasta Apps User.php:**

PHP ARTISAN MAKE:MODEL PRODUCT (NO SINGULAR)

PHP ARTISAN MAKE:MODEL STORE

* Ele vai criar na pasta app, um arquivo chamado **Product.php**
* Vou crie todos os arquivos necessários para o Model com **Store.php**



**12° ORM (Objetc Relation Mapping):**

ORM e um cara que traduz, uma base relacional que e o nosso banco de dados, para um ponto que e Orientado a Objetos (POO) que e os nossos **Models.**

**NOTA:** O Laravel, ele tem uma convenção pra entender qual tabela esta associado com esse Model, ele vai procurar seu model no plural automaticamente lá no **Banco de Dados**

* Então se ele foi criado store, ele vai procurar automaticamente Stores, lá na base de Dados.
* Então se ele foi criado como Products, ele vai procurar automaticamente por Products, lá na base de Dados.

**ROUTES**

**13° Routes: Acessando a Pasta Routes, no arquivo web.php**

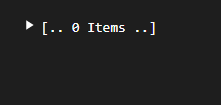
Definindo uma rota básica: Route::get(‘/model’, function (){

$product = \App\Product::all(); //SELECT \* FROM Products;

Return Product;

});

**Retornando isso, podemos visualizar esta rota no navegador 127.0.0.1:8000 o retorno de um array vazio, pois não temos nada na tabela products nenhum dado inserido com seed.**



**Array vazio;**

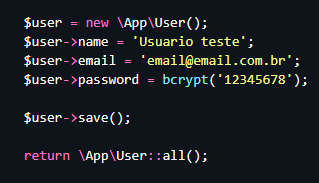
**ACTIVE RECORD**

**14° Active Record o que seria?**

Padrão de Projeto que chamamos de active record criaremos alguns dados!

O active Record permite trabalhar as colunas do banco como se fosse atributos da minha classe, existe uma função do Laravel chamada bcrypt () que encrypta o password

**NOTA: Tem como eu utilizar o método find() que pega o usuário pelo ID.**



**Querys e Eloquent:**

Porque que quando utilizamos o método all com return ele retorna um Json pra nós?

O Laravel quando retornamos um objeto all, ele serializa isto pra gente, ou seja, ele pega a estrutura do objeto e converte pra **Json.**

O método All retorna o Objeto chamado de Colection:

Essa Colection nos permite fazer diversas operações em cima de uma coleção de dados vindas do banco.

* Posso chamar todos os usuários chamando o método **all**() return \App\User::all()
* Posso buscar um único usuário chamando o método **find** (3) return \App\User::find (3)
* Posso colocar uma condicional chamando o Where, **Where**(name, nomeUsuario)->get

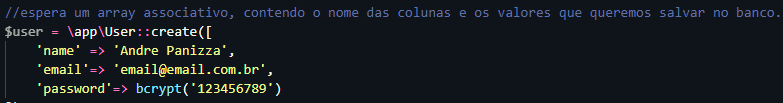
**Nota:** Quando utilizo o método Where, eu já preciso passar o ->GET() recuperar;

* Posso buscar também o primeiro resultado com o método first com Where()->first();
* Posso retornar um resultado Paginado, com o método Paginate();

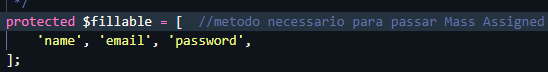
**MASS ASSINGNED OU MASS UPDATE**

Atribuição em Massa: Mass Assingned.

A função create espera receber uma array associativo com os dados da minha tabela, que vão ser inserido no banco de dados.



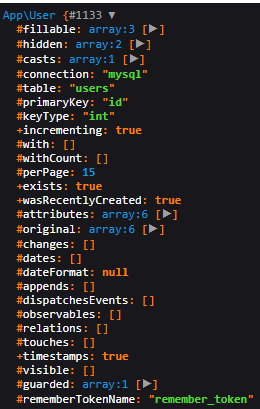
Create, precisa que o model, tenha um atributo chamado Filable, que está dentro de (Model) **User.PHP**



Nota: Se comentar as linhas desse atributo protegido **$fileable**, temos um erro.

Ele me diz quais campos eu posso receber valores, quando eu utilizar **mass assingned**

**Utilizando um método dd (dump die) retornando a criação:**

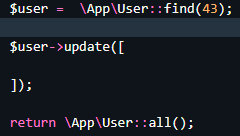


NOTA: Ele retornou um Objeto do tipo User.

**MASS UPDATE**

**Atualizando o usuário 43, que foi inserido com mass assingned!**

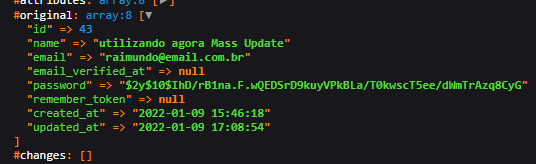
**Dessa forma abaixo, mantenho os dados do usuário que eu busquei:**



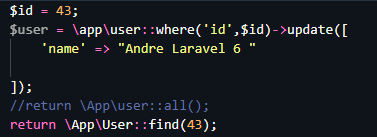
**Passo um array com o método update()**

**NOTA: diferente do create, que já retorna o objeto que foi inserido no banco,**

**O Update retorna True ou False; O update retorna um bool.**

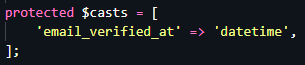
**ALTERAÇÃO DOS DADOS COM UPDATE**

**Outra forma de usar o Update:**



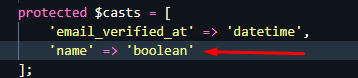
**15° ENTENDENDO CATS E HIDDEN**

O atributo Casts, ele faz a conversão pra gente automática, no campo em questão por exemplo:



O email\_verified\_at ele seja do tipo Datetime

**Por Exemplo:** Quando eu utilizar o name, ele seja toda vez convertido em bolleano.

****

**No resultado todo name, será um bolleano agora:**

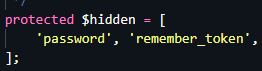


Então todas as colunas que estiverem definadas no casts do tipo definido pra elas, será convertido automaticamente quando começarmos a trabalhar com nossos models, fazendo buscas através dos models.~

Então o casts permite ir na coluna ali nos models pra converter o tipo de dados das colunas, converter um valor, para um valor que você espera.

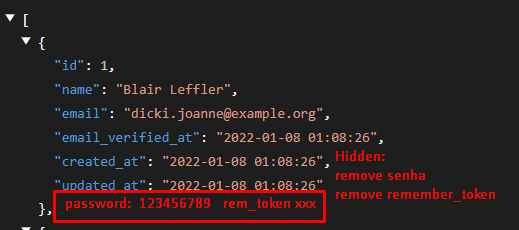
**Hidden:**

O que o atributo Hidden faz:



* Quando fizermos as buscas pelos nosso dados, e retornar pra nos a Collection,

Ele automaticamente, remove o campo senha, da listagem:

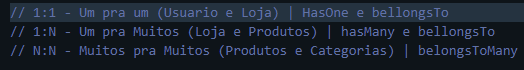


Para ocultarmos alguns dados, e só listarmos os nomes das colunas que ele não exibira em nossa collection!!

Então esta e a Intenção da propriedade Hidden.

# **ELOQUENT E OS RELACIONAMENTOS**

O eloquent possui alguns métodos que representa algumas ligações e relacionamentos das lojas:



Esses 03 métodos via Eloquent.

# **CRIANDO AS RELAÇÕES E OS METODOS VIA ELOQUENT**

Vamos criar o método que representam esta ligação, por exemplo se eu estou em User

Então vou criar um método stores no plural, e vamos retornar o método que representa esta ligação, neste momento o **User:**

Public function store( ) {

Return $this->hasOne(Store::class);

}

* Temos que passar qual e o objeto ou model que esta associado, lembrando que

hasOne: (Tem Um)

bellongsTo (Um pra Muitos)

ToMany (Produtos e Categorias)

Precisamos dizer qual o objeto, ou Model, ou classe que esta associada nesta ligação com este usuário e quem que vai ser **Store:**

Public function User( ) {

Return $this->belongTo(User::class );

}

Agora a relação com os **PRODUCTS**

* **Dentro do Store:** Vamos passar uma função:

**Public function product() {**

**Return $this-> ToMany(Product::class)**

**}**

Então Note que temos:

Um método que retorna o relacionamento com Users:

**Public function User() {**

**Return $this->belongTo(User::Class)**

**}**

**De 1 pra N**

E temos um relacionamento de Products para Store dentro de Store:

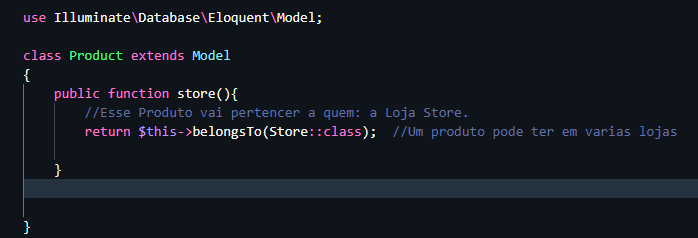
**Public function Product() {**

**Return $this->ToMany(Product::class)**

**}**

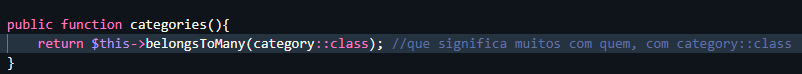
Mapeando Agora a relação do ponto de vista dos Models:

No arquivo Product:



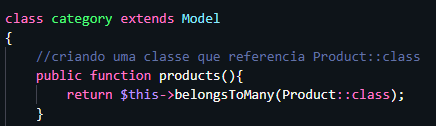


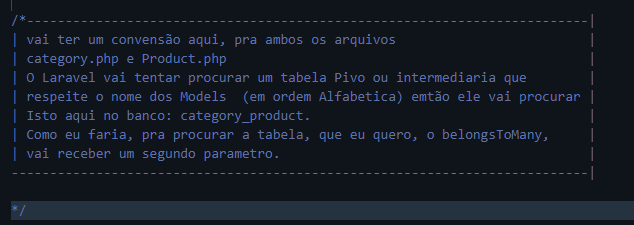
Vou ter o Método Categories:



Que significa muitos com quem, com a classe ou objeto category.

E vou ter o Método em Category:





# **MODEL E FACTORYS PARA TESTES**

**16° Vamos trabalhar agora com as Factorys, lembrando que elas nos auxiliam na criação das nossas seeds, para criarmos dados em massa, com o método faker.**

Então criaremos os 03 arquivos:

- PHP ARTISAN MAKE:FACTORY StoreFactory



- PHP ARTISAN MAKE:FACTORY ProductFactory



- PHP ARTISAN MAKE:FACTORY CategoryFactory



Depois desses três arquivos criados, vamos criar dados, utilizando o método faker, para testarmos inserções no banco de dados:

Lembra-te que temos as tabelas (migrations) com algumas informações inseridas no banco de dados:



Contendo as seguintes colunas:

**$table->bigIncrements(id),**

**$table->unsignedBigInteger(user\_id),**

**$table->string(name),**

**$table->string(description),**

**$table->string(phone),**

**$table->string(mobile\_phone),**

**$table->string(slug),**

**$table->timestamps();**

**$table->foreign(user\_id)->references(id)->on(users)**

**ESSAS COLUNAS ESTÃO DECLARADAS NO BANCO, PORÉM SEM VALORES NENHUM!**



Contendo as seguintes colunas:

**$table->bigIncrements(id),**

**$table->unsignedBigInteger(store\_id),**

**$table->string(name),**

**$table->string(description),**

**$table->text(body),**

**$table->decimal(price, 10,2),**

**$table->timestamps();**

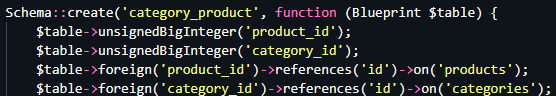
**$table->foreign(store\_id)->references(id)->on(stores)**

As factorys serão preenchidas com dados $faker que serão usados para testarmos a inserção dos dados no Seeds que já foram criados.

NOTA: Existem mais 02 Migrations que foram criadas somente para fazer referencias de categorias de produtos:

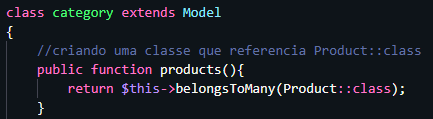


Que contém:



**NOTA:** Que temos o Model que faz a relação que referencia Products:

MODEL:



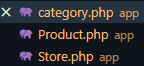
AGORA COM OS 03 FACTORYS DEVIDAMENTE PREENCHIDOS, COM O $FAKER-> CHAMADO CADA METODO, VAMOS UTILIZAR O METODO MASS ASSINGNED PARA CRIARMOS. PRECISAMOS DEFINIR O

**$fillable** = [“E o método necessário para passar o MASS ASSINGNED”] sem ele não é possível.

**$hidden** = [“define qual coluna deve ficar oculto se passarmos aqui”]

**$casts** = [“Define o tipo de dado que é retornado automaticamente pra nossa collection”];

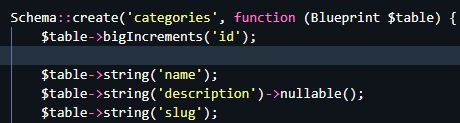
**17° Dentro dos** **Models:**

 ***Precisamos definir os*** $**FILLABLE = [ ];**

1° Category.php Model vai receber:

**Protected $fillable = [NAME, DESCRIPTION, SLUG] porque:**

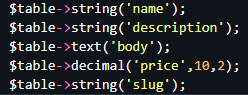
**Pois na minha Migration eu tenho os campos:**



2° Product.php Model vai receber:

**Protected $fillable = [NAME, DESCRIPTION, BODY, PRICE, SLUG] porque:**

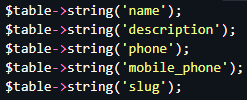
**Pois na minha Migration eu tenho os campos:**



3° Store.php Model vai receber:

**Protected $fillable = [NAME, DESCRIPTION, PHONE, MOBILE\_PHONE, SLUG ]**

**Pois na minha Migration eu tenho os campos:**



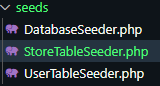
**Agora vamos gerar nossas SEEDS Loja e os Produtos.**

**18° primeiramente antes de tudo vamos criar um arquivo Seeder:**

PHP ARTISAN MAKE:SEEDER STORETABLESEEDER



AQUI EM SEEDS:



Vamos fazer uma alteração no UserTableSeeder:



Temos os 40 usuários, com base na definição da factory do Model user, e criamos isso utilizando o método create;

Podemos chamar um método chamado de each () ou seja (cada).

Para cada uma dessas execuções, podemos fazer a criação para cada um desses usuários.

Então vou receber um parâmetro User, que vai receber o usuário criado, ai eu posso receber a minha ligação store.

Ai eu posso chamar a minha ligação store() e chamar o método create

Lembre-se: store, representa a nossa ligação de 1 pra 1 hasOne de usuário e a loja.

Posso chamar uma factory, passando o namespace(\APP\Store::Class)->**MAKE**

**O make:** O make, ele vai fazer o que ele vai criar um objeto store com as informações Fake.

Precisamos usar o método Save = [Trabalha com objetos]

O método Create = [Trabalha com Arrays], por isso da Mudança para criarmos os objetos:

## COMANDO:

PHP ARTISAN MIGRATE:FRESH

Para zera tudo de fato: Agora nosso banco de dados está zerado.

PHP ARTISAN DB:SEED



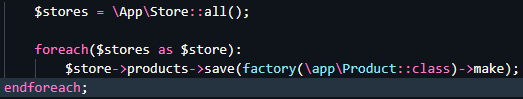
Então ele criou 40 users e 40 stores uma para cada usuário.

**19° Agora vamos inserir valores dentro dos Products, como podemos fazer isto:**

Dentro de StoreTableSeeder, chamo o método $store = \App\Store (QUE E O MODEL) passando o método all:



Para referenciar, vou Pegar a minhas Lojas “stores” passando um looping foreach:



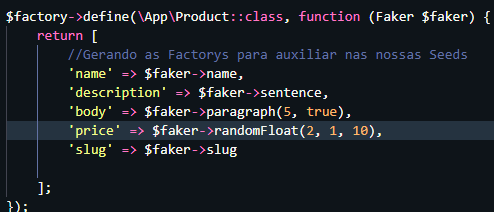
Chamando a ligação de store com o products, chamando o método Save( ) e dentro de save, chamo a minha factory pro model Product fazendo um make.

Vai adicionar um produto para cada loja que eu tenho, fabricará o objeto produto.

**Rodando a seed individualmente:**

PHP ARTISAN DB:SEED –CLASS=STORETABLESEEDER

NOTA: verificar o ProductFactory:



Alterar o campo do método randomFloat().

**Depois que gerar essa Seed, temos no campo Products, 40 produtos associado as Lojas 40 Lojas!**

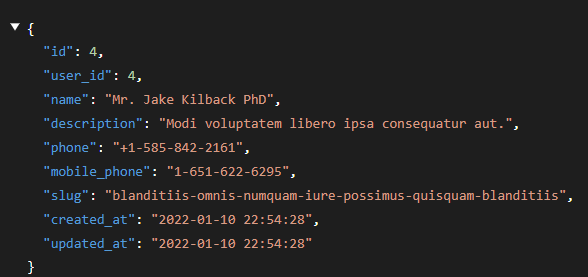
## **QUERIES COM RELAÇÕES**

20° Como trabalhar com os Models:

Como eu faria para pegar a loja de 1 usuário, posso chamar da seguinte maneira, posso chamar um $user = \App\User::find(4) utilizando o método Find

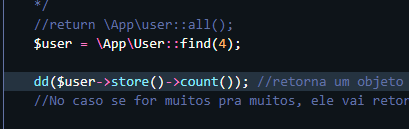
Vamos retornar com return $user->store;

Note que ele vai trazer pra mim a Loja 4:



Ele retorna um objeto único, neste caso qual foi o Objeto único, neste caso foi o store.

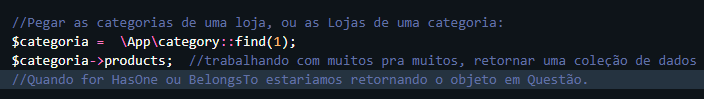
Vai retornar pra mim uma instancia de HasOne! Neste caso posso dar um count:



Que será retornado:

Consigo fazer um count() apartir do atributo, pois retorna uma Collection.

//Pegar as lojas de uma categoria de uma Loja?



**INSERINDO RELAÇÕES:**

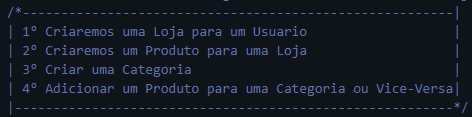
21° Vamos conhecer como podemos criar alguns dados em cima das relações dos nossos Models:

//**Criaremos uma Loja para um Usuario**

//**Criaremos um produto para uma Loja**

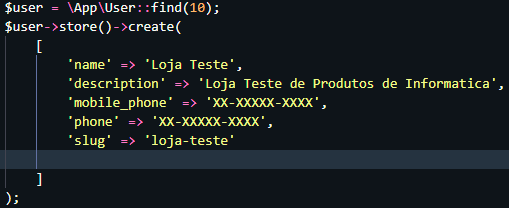
//**Criar uma Categoria**

//**Adicionar um produto para uma Categoria ou Vice e Versa.**



* Primeiro Buscamos o usuário na nossa base utilizando o método Find()
* $user = \App\User::find(10);
* Agora vou acessar a Ligação: chamar o método create() que lá na factory usamos:

Vamos utilizar o método **Create** pra criarmos uma **Loja,** que vai receber a referencia do usuário desse **$user = \App\User::find(10)**

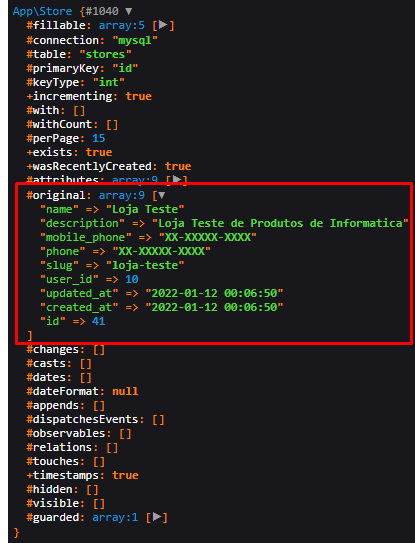


NOTA: como estou acessando a ligação e criando uma loja em cima de um relacionamento, o retorno vai ser um objeto com essas informações, e o ID recebido na base.

Então se fizermos um DD no Objeto Store:

Temos a criação dessa loja com o ID 41.

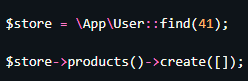
**Reforçando:** O create sempre vai retornar o objeto, já com o ID que ele recebeu do banco de Dados.



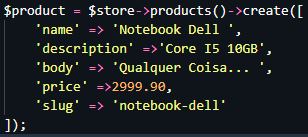
**Nossa Inserção já foi feita aqui, com sucesso, bem tranquila e bem Direta.**

**Agora como criar um produto pra Loja:**

Vou chamar $store->products()->create() e o método create.

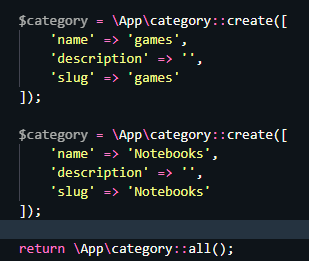


**Esse método create usaremos como base o ProductFactory, para inserir os dados, só para agilizar:**

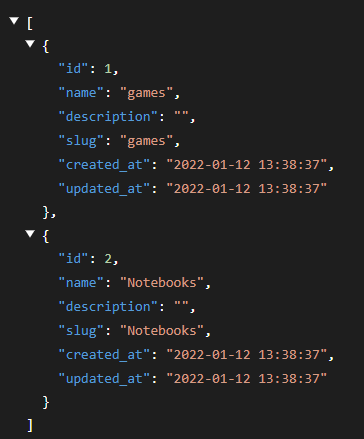


## CRIANDO UMA CATEGORIA:

Vamos criar 02 categorias com o Model category, e vamos retornar todos os objetos serializados:



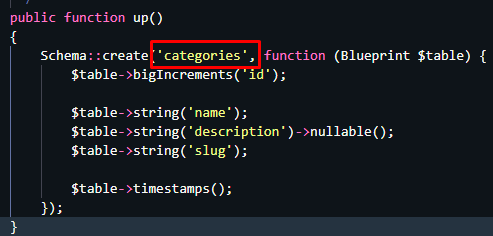
Resultado seria este:



**Como e que eu faço, para um produto adicionar uma dessas categorias?**

**$product = \App\Product::find(49) NOTA: aqui estamos acessando nosso model direto.**

$product->categories // aqui estamos acessando a tabela de Migration de nome tabela (categories) que tem essas informações:



Então:

$product->categories->attach();

O attach o que ele faz, ele pega e vai adicionar pra gente, os Ids que passarmos aqui referente as categorias:

Vou Adicionar para esse produto (49), uma categoria para ID (1);

Vamos fazer um dd() desse cara pra vermos o retorno deste attach [adiciona] ligações

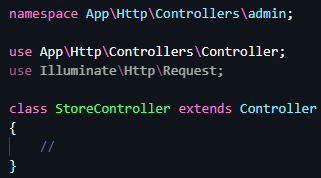
Se você Quiser remover os Ids que estão na Base, teremos o Detach [remove] ligações

O sync ele e interessante, porque ele vai fazer o seguinte: sync adiciona todos os itens.

## Rotas, controllers, e criação dos Cruds

22° Como gerar os controllers:

PHP ARTISAN MAKE:CONTROLLER ADMIN/STORECONTROLLER



**Criou uma pasta admin, dentro da pasta controller, já adicionou o namespace para gente.**

Nossos Cruds, sempre estarão dentro da pasta Admin, os controllers referentes a nosso painel.

# ROTAS WEB:

23° Iniciando com Rotas:

O Laravel, vai responder diretamente para as Rotas:

//Route::get

//Route::post

//Route::put

//Route::patch

//Route::delete

//Route::options

//Então esse método que vemos aqui em cima GET, recebendo um parâmetro, nome da Rota, e um callback no segundo parâmetro

//serve pra qualquer um desses aqui abaixo ou seja:

/\*

1° Get eu recupero as coisas

2° Post eu crio alguma coisa

3° Put Atualização

4° Patch Atualizações

5° Delete Remoção

6° Options ele me retorna quais cabeçalhos aquela rota especifica ela responde.

\*/

Vamos começar com as rotas:



INICIANDO COM VIEWS:

24° Vamos criar uma pasta admin dentro da pasta View, uma pasta stores, e dentro desta pasta vou criar um arquivo index.blade.php, aproveitar e mandar o stores pra ela:



Nota, dentro do arquivo index.blade.php:

* Teremos que usar a Anotação do Laravel que se é semelhante com a do PHP:

E aqui dentro eu colo a TR que vai receber o looping, ou o laço em cima dela:

@foreach(‘stores’ as store)

 <tr>

        <td></td>

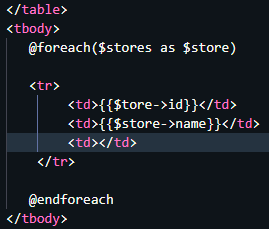
        <td></td>

        <td></td>

    </tr>

@endforeach

E pra cada coluna eu pego:



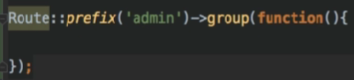
Atualizando Teremos: o resultado Abaixo:

**TEREMOS ESTE RESULTADO AQUI**:



CONFIGURAÇÕES DE ROTAS

Podemos passar um prefixo para as Rotas:

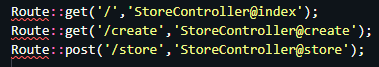


**Vai receber um grupo de Rotas:**

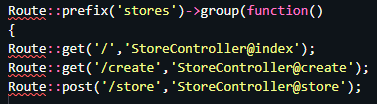
Eliminando assim o conceito de admin:



E colocar o meu grupo dentro:

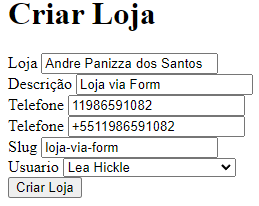


Podemos melhorar mais ainda passando outro prefixo agora stores dentro:

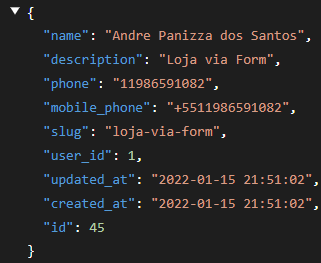


Se acessarmos as Rotas temos os resultados!

[127.0.0.1:8000/admin/stores/create](http://127.0.0.1:8000/admin/stores/create)



[127.0.0.1:8000/admin/stores/store](http://127.0.0.1:8000/admin/stores/store)



Vamos simplificar mais ainda esta parte de Rotas, utilizando o prefixo e o Namespace, fazendo isso de rotas e funções recursivas!

USANDO TEMPLATES E BLADE

**25° Melhorando ainda mais as Rotas:**

Vamos agora criar dentro de Views uma pasta chamada Layout, dentro dela vamos criar um arquivo chamado Layout.php

Aqui vai ser nosso Layout base do nosso Administrativo, deixando o que é repetido.

Podemos definir um ponto, no layout base onde será utilizado em todas as Views, uma área que vai receber o conteúdo da View em questão:

Nos chamamos isso de **Herança de Templates**.

Como defino está área:

 @yield('content')

Como eu faço pro create, usar o app.blade quando eu carregar ele.

Utilizando a notação @extends(layout.app) e como se tivesse chamando a pasta layout/app.blade.php ele herda e busca este arquivo, que tudo que estiver dentro do content, irá para o @yield(contente).

NOTA: Section e uma diretiva do blade.

A anotação @section e @endsection, faz a abertura da seção e o fechamento dela.

**26° Criando Edição:**

Adicionando parâmetro dinâmico na ROUTE, store:

Route::get('/{store}/edit','StoreController@edit');

E vamos mandar isso para um Update/store:

Route::post('/update/{store}','StoreController@update');

Neste caso, este update aqui vai ser o processamento da dessa nossa loja.

Aqui no caso, estou passando o parâmetro {store} dinâmico na minha URL, que vai ser o ID da loja, a mesma coisa estou fazendo pro Update.

**Como e que eu intercepto isso no meu controller:**

Primeiramente vamos criar outros Métodos:



O que vier da URL será passado parâmetro $store do edit();

        $store = \App\User::find();

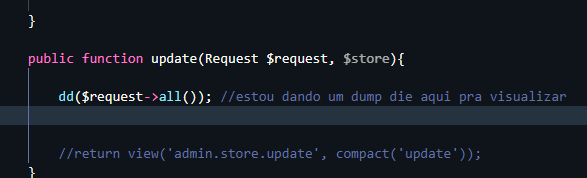
Posso passar o parâmetro com o caminho utilizando um cara que já conhecemos que e o Find.

E passar o parâmetro store para ele();

Com isso eu retorno quem:

        return view('admin.stores.edit', compact('store'));

**No caso do Update vou fazer a mesma coisa:**

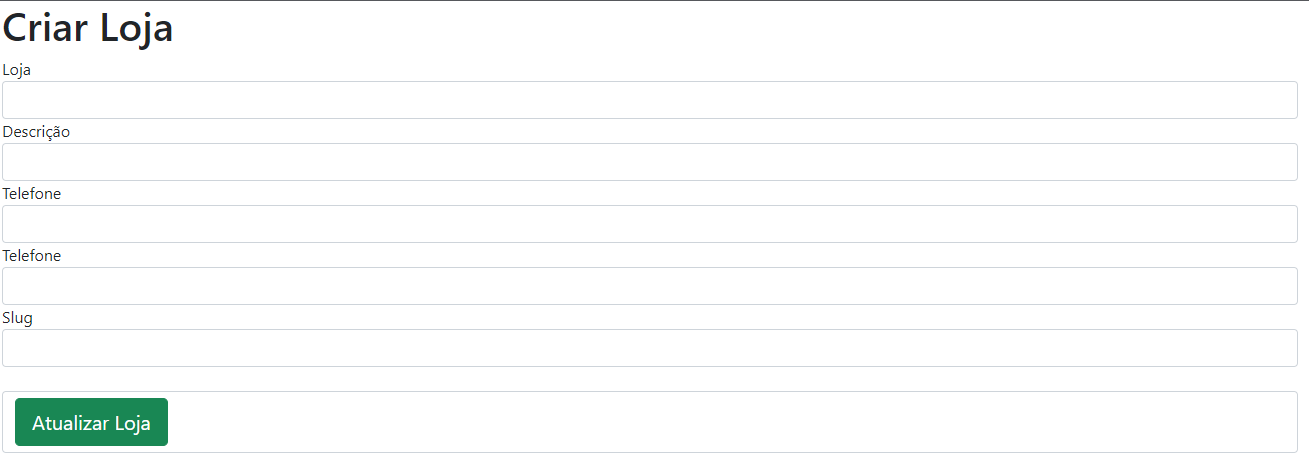


Dando um Dump Die, pra visualizar.

Depois criamos o nosso edit.blade.PHP copiando e colando.

Nosso Layout ficara desta forma:

**Acessando 127.0.1:8000/admin/stores/11/edit , note que o parametro dinamico funciona!**



Pra atualizar, vamos excluir o dd() Dump Die, e chamar a variável $data.

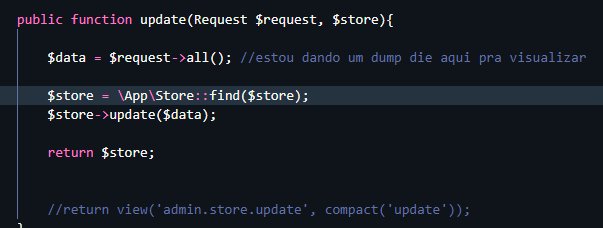
Então $data = $request->all() //método all

$store = \App\store::find($store);

$store->update($data);

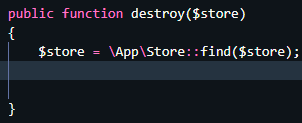
Return $store;

* Lembrando que o retorno do Update e um Booleano, já o retorno do create, vai ser o próprio objeto com a referência recebida do banco.



**27° Método de Remoção:**

Vamos criar um Método, para criar a logica de remoção de 1 item no banco de dados:

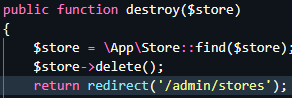


Esse Cara vai receber um parâmetro da URL, passando um store, agente precisa buscar o dado no banco:



Passando o parâmetro dinâmico que vem da URL!

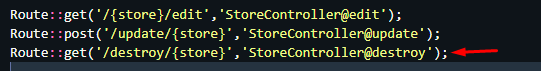
Chamando o Método DELETE dessa forma:



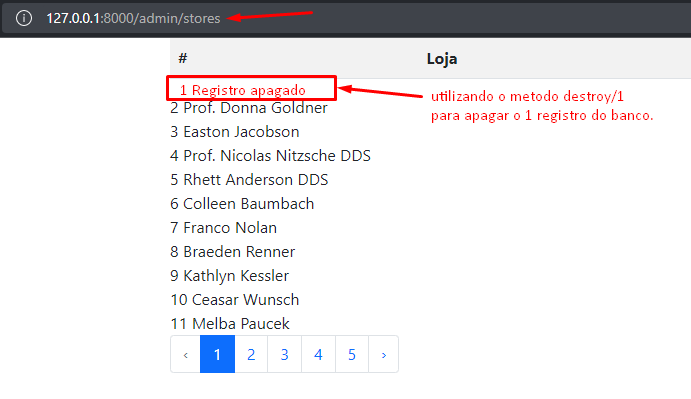
**NOTA: Depois que eu fizer o Delete, estou redirecionando para /admin/stores/**

Feito isso eu preciso Linkar, criar a Route que apontara para esse Método

No web Routes:



**Crio a minha rota, do destroy com o parâmetro dinâmico.**



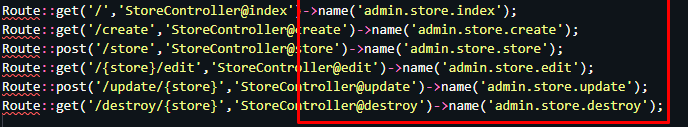
**Destruído com sucesso.**

**Agora vamos Linkar os botões de Criar, remover, e Atualizar os dados:**

## FAZENDO AS MELHORIAS DE LINKAGEM DOS BOTÕES:

Temos a possibilidade de definir apelidos para cada uma das nossas rotas, como isso funciona:

Chamaremos cada uma de nossas rotas um método chamado Name.



Pronto, redefinimos as Rotas, agora como vou colocar meu apelido lá no meu edit.blade.php, e no meu create.blade.php

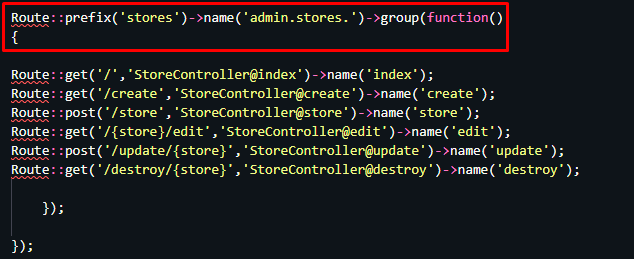
Agora vamos chamar uma função nomeada como Route, esse Route espera o apelido da Rota:

<form action="{{route('admin.stores.store')}}" method="POST" enctype="">

Note, que agora, estou chamando o nome do apelido, não o nome da Rota Real, e uma boa pratica sempre referenciar pelos apelidos deles, e os apelidos sempre pode referenciar pelo esse método name

## **DEFININDO APELIDO BASE PARA O GRUPO DE ROUTAS:**

Dessa forma simplificamos ainda mais o grupo de Rotas passando o admin.stores



Podemos colocar um name também no Group, após o namespaces:

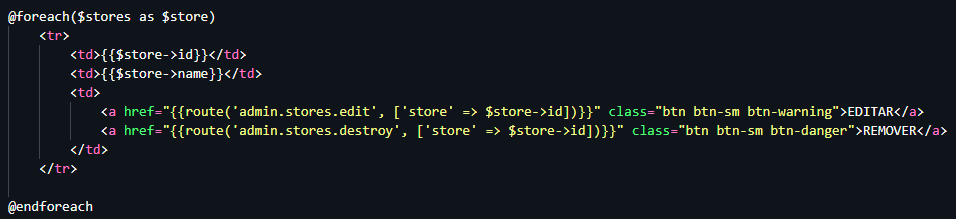
Route::prefix('admin')->name('admin.')->namespace('admin')->group(function()

Assim alterando o name da route abaixo:

Route::prefix('stores')->name('stores.')->group(function()

## **Agora vamos linkar e criar os botões no create.blade.php e no edit.blade.php**

Vamos ao Nosso arquivo index, que temos em cada coluna uma action



Depois colocaremos um botão de Criação de Loja:

<a href="{{route('admin.stores.create')}}" class="btn btn-lg btn-success">Criar Loja</a>

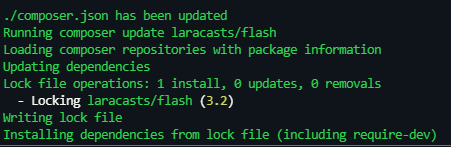
# **Definindo Mensagens Flash**

Vamos trabalhar com pacotes, para o usuário, o nome do pacote no github.com/laracasts/flash.

Laracasts/flash, ele usa as classes do Bootstrap, e ele e bem simples para utilizar mensagens de execução!

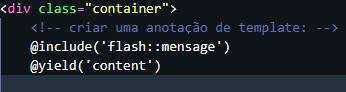
Então o Primeiro passo:

No nosso terminal usaremos: composer req laracasts/flash para instalar:



Depois de Instalado, o que precisamos fazer, aqui no nosso layout app.blade.php, antes do Yield, iremos fazer a chamada de um arquivo vai vir de dentro desse pacote flash.

Para passar a anotação do Include(‘’) para anotação do arquivo:



Como faço pra mandar a mensagem e ela ser exibida, la no nosso store.Controller

Lá no nosso retorno do método update, podemos chamar flash(‘Loja atualizada com sucesso’)->success(); chamo o método success pra vir na cor verde.

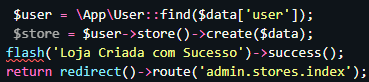
E depois eu retorno e dou um redirect para rota que eu quero!

A boa pratica e redirecionarmos para apelidos do que a Rota real ok, então chamamos um método chamado route.



**Atualizaremos todos:**





Modulo de crud de Produtos:

**28° Vamos começar gerando os controllers dos produtos com o Comando:**

Php artisan make:Controller Admin/ProductController --resource

Vou criar este controller dentro da pasta Admin vai pegar o namespace admin também,

Vou gerar este cara, Note que ele trará, métodos já criados pra gente, o professor irá criar cada um dos nossos Métodos.

Controllers como Recurso, o que são:

Esse cara realmente e chamado de controller como recurso, ele já traz pra gente uma serie de métodos semiprontos

1° Este e um Método pra Listar os meus dados

2° O create aqui, e pra exibir o formulário de Criação

3° O store, ele já vai fazer o processamento da criação

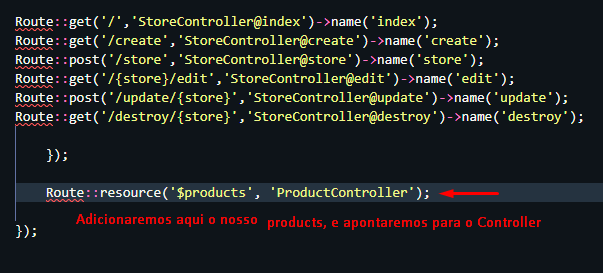
4° O Show, caso eu queira criar uma visualização rápida de um Dado, posso implementar dentro do método Show

5° O método edit, posso exibir o formulário de Edição certo.

6° Temos aqui o Update, pra fazer o processamento da Atualização

7° E tenho o Método destroy, que servirá para fazer a remoção, dos campos.

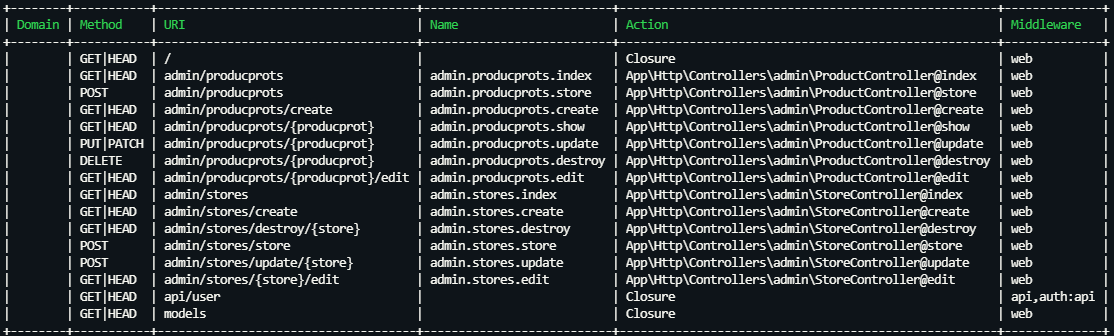
Vamos no nosso ROUTES:



Vamos fazer um comando:

**PHP artisan Route:List** pra vermos todas as Rotas do nosso projeto.

E ai explicaremos, o que o método resource do Route faz pra gente:



Vamos fazer o nosso crud utilizando o Controller como recurso, fazemos a alteração lá na parte de Lojas:

Começando o nosso código com Cruds:

No Product controller, essa e uma nova formula de trabalhar e relacionar o seu model com o controller, vamos definir primeiro o nosso construtor:

