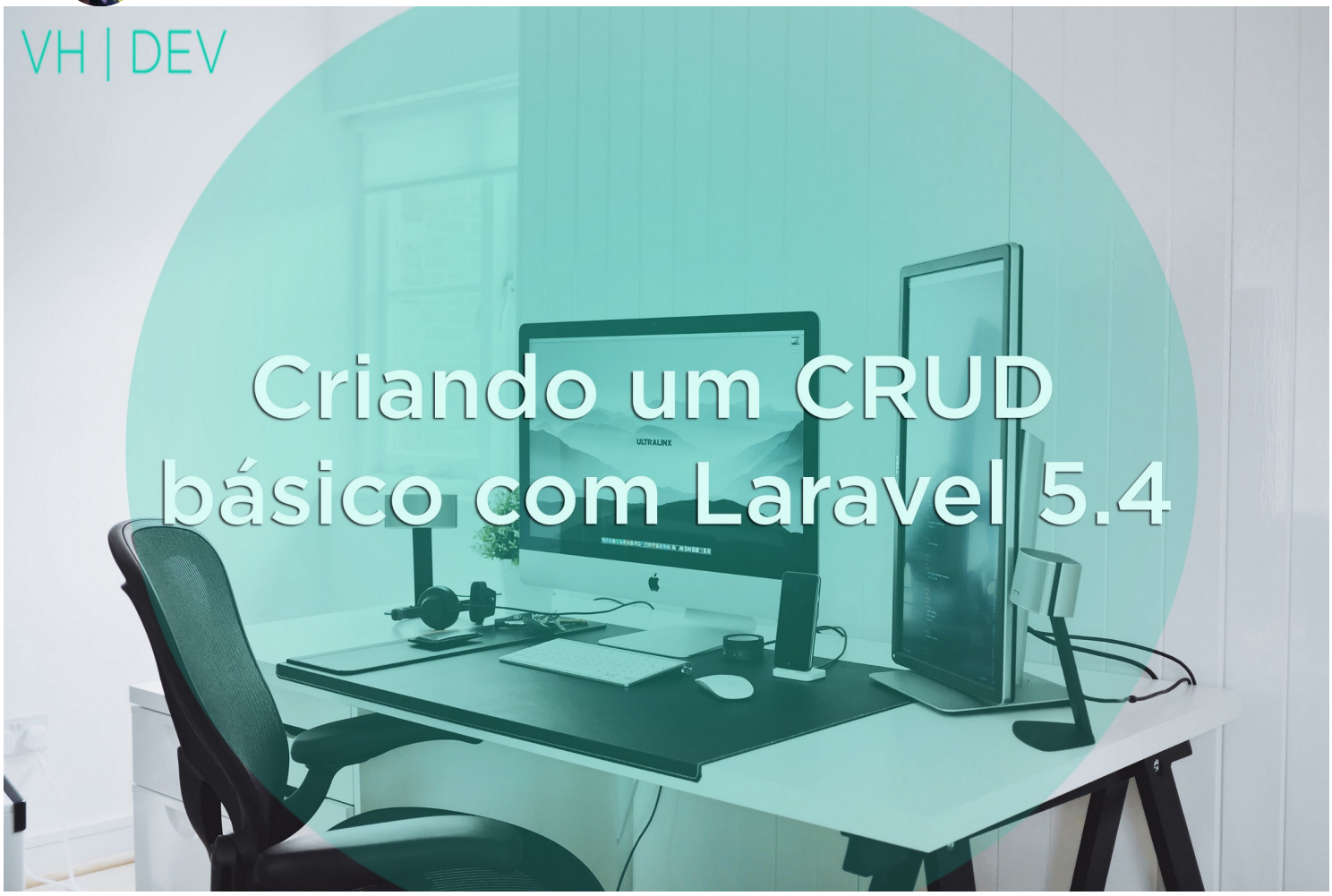
Criando um CRUD básico com Laravel 5.4

Victor Hugo Rocha in Training Center
May 7, 2017 · 4 min read



Olá, DevZ.
Nesse post eu vou partir do princípio de que você já sabe começar o seu projeto em Laravel e está querendo dar os seus primeiros passos com desenvolvimento web. Algum tempo atrás eu escrevi dois textos um sobre Migrations e outro sobre Faker & Factory que podem complementar o conteúdo dessa publicação.
O Laravel é um framework MVC, **Model-View-Controller**, que é um dos vários padrões de design de software, temos também o MVW, MVVM mas vamos nos focar no padrão MVC que possuem como fundamento a separação de conceitos e a reusabilidade de código dentro do projeto. Esse é

Vamos começar pela migration. Como você já sabe ela é como um controle de versão do seu banco de dados e permite que você adicione tabelas e colunas no seu banco sem muita dificuldade. Para criar a sua migration você deve rodar o comando *artisan* abaixo.

```
php artisan make:migration create_nametable_table
```

```php
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateProductsTable extends Migration
{
    public function up()
    {
        Schema::create('products', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name', 30);
            $table->text('description')->nullable();
            $table->string('quantity');
            $table->decimal('price', 5, 2);
            $table->timestamps();
            $table->softDeletes();
        });
    }

    public function down()
    {
        Schema::drop('products');
    }
}
```

create_products_table.php hosted with ♡ by GitHub view raw

php artisan make:migration create_products_table

Na function **up** eu estou criando uma tabela *products* com os campos id, name, description, quantity e price. Eu explico o que são os campos *timestamps e softDeletes* aqui. A função **down** dropa a tabela

Agora vamos falar da Model. O Laravel possui integrado o Eloquent ORM que é uma forma mais bonita e abstrata de interagir com o seu banco de dados. Cada tabela de banco de dados tem um *Model* correspondente que é usado para interagir com essa tabela. Para criar a sua Model basta rodar o seguinte comando no terminal. Se você passar a Flag -m é criada uma migration referente a Model.

```
php artisan make:model NameModel
php artisan make:model NameModel -m
```

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Product extends Model
{
    protected $fillable = ['name','description','quantity','price'];
    protected $guarded = ['id', 'created_at', 'update_at'];
    protected $table = 'products';
}
```

Product.php hosted with ♡ by GitHub view raw

No meu caso a minha model Product é bem simples e possui três variáveis protegidas: *fillable*, *guarded* e *table*. A variável fillable define quais os campos que podem ser inseridos pelo usuário do sistema no Banco, o campo guarded protege os campos de inserções. Ele impede que alguém insira dados em alguns campos da nossa tabela.
Mesmo que não seja possível realizar essas inserções através do Front-end é importante que de alguma forma protejamos esses campos de possíveis vulnerabilidades que possam ser explorados por usuários mal intencionados. Agora que definimos na Model qual campos podem ser preenchidos pelo usuário podemos trabalhar na nossa **CRUD**.
Se você gostou do Eloquent você também pode usá-lo em um projeto que não seja Laravel. Você pode encontrar mais informações sobre isso na documentação.

php artisan make:controller NameController

```php
<?php

namespace App\Http\Controllers;

use App\Http\Requests\ProductRequest;
use App\Product;

class ProductController extends Controller
{
    public function index()
    {
        $products = Product::orderBy('created_at', 'desc')->paginate(10);
        return view('products.index',['products' => $products]);
    }

    public function create()
    {
        return view('products.create');
    }

    public function store(ProductRequest $request)
    {
        $product = new Product;
        $product->name        = $request->name;
        $product->description = $request->description;
        $product->quantity    = $request->quantity;
        $product->price       = $request->price;
        $product->save();
        return redirect()->route('products.index')->with('message', 'Product created successfull
    }

    public function show($id)
    {
        //
    }

    public function edit($id)
    {
```

```
    {
        $product = Product::findOrFail($id);
        $product->name        = $request->name;
        $product->description = $request->description;
        $product->quantity    = $request->quantity;
        $product->price       = $request->price;
        $product->save();
        return redirect()->route('products.index')->with('message', 'Product updated successfull
    }

    public function destroy($id)
    {
        $product = Product::findOrFail($id);
        $product->delete();
        return redirect()->route('products.index')->with('alert-success','Product hasbeen delete
    }
}
```

ProductController.php hosted with ♡ by GitHub view raw

Dentro da Controller você pode ver que criamos as funções referentes ao que uma CRUD deve fazer. Essas funções devem ser chamadas nas rotas como métodos da sua controller. Há duas formas de se fazer isso.

```
<?php
...

Route::group(['prefix' => 'admin', 'middleware' => 'auth'], function () {
    Route::get('/', 'AdminController@getIndex');
    Route::get('projetos', 'ProjetosController@getIndex');
    Route::get('projetos/inserir', 'ProjetosController@getInserir');
    Route::post('projetos/inserir', 'ProjetosController@postInserir');
    Route::get('projetos/editar/{id}', 'ProjetosController@getEditar');
    Route::post('projetos/editar/{id}', 'ProjetosController@postEditar');
    Route::post('projetos/deletar/{id}', 'ProjetosController@postDeletar');
...
});

?>
```

route.php hosted with ♡ by GitHub view raw

fazer dessa maneira mas na versão 5.4 e superiores há um facilitador do Laravel, você pode fazer as mesmas 9 linhas do código anterior em apenas uma.

```php
<?php
...

Route::resource('products', 'ProductController')->middleware('auth');

...
?>
```

route.php hosted with ♡ by GitHub view raw

Com essa declaração de rota única você deixa para que o Laravel crie todas as rotas para a variedade de ações do *ProductController, dessa maneira, diferente do exemplo anterior, ele define os verbs PUT/PATCH* para *Update* e *DELETE* para *Delete* da sua CRUD de Produtos.
Você também pode fazer isso direto do terminal caso crie o controller dessa maneira. Note que além de definir que aquele é um Resource Controller dessa maneira você também indica qual o Model correspondente aquele Controller.

```
php artisan make:controller NameController --resource --
model=NameModel
```

. . .

Se você chegou até aqui não esqueça de deixar o seu ♥ e/ou um comentário. I ♥ seu Feedback. Se você tem interesse por Bots ou gostou do meu texto você pode ler algum dos meus outros artigos, recomendo:

1. Telegram Bot — Como desenvolver seu Bot com Javascript e Heroku
2. Docker — Dockerhub, pull e push nas suas imagens
3. TDD — Test Driven Development

See ya!

Laravel Web Development Mvc PHP Development

Training Center

Conectamos pessoas que querem aprender algo relacionado a desenvolvimento de software com gente que pode guiá-las.

Follow

See responses (10)

More From Medium

Re-imagining Education through Blockchain

Puneet Goenka in...
Aug 4, 2018 · 4 mi...

Understanding the Blockchain

Cher Yi
Nov 15, 2017 · 12...

Explain machine learning to someone who has made a “hello world” (pt. 1)

BT
Jun 14, 2018 · 11 m...