Faculdade de Tecnologia de Americana "Ministro Ralph Biasi"
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Marcelo Leandro de Araujo Junior

Desafio de Desenvolvimento de uma Aplicação Laravel

Americana, SP

2021

Faculdade de Tecnologia de Americana “Ministro Ralph Biasi”

Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Marcelo Leandro de Araujo Junior

Desafio de Desenvolvimento de uma Aplicação Laravel

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, sob a orientação do Prof. Esp. Antonio Alfredo Lacerda.

Área de concentração: Programação.

Americana, SP

2021

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

A689d ARAÚJO JÚNIOR, Marcelo Leandro de

Desafio de desenvolvimento de uma aplicação Laravel. / Marcelo Leandro de Araújo Júnior. – Americana, 2021.
57f.

Monografia (Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas) - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza

Orientadores: Prof. Antonio Alfredo Lacerda

1 Desenvolvimento de software I. LACERDA, Antonio Alfredo II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana

CDU: 681.3.05

Marcelo Leandro de Araujo Junior

Desafio de Desenvolvimento de uma Aplicação Laravel

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, sob a orientação do Prof. Esp. Antonio Alfredo Lacerda.

Área de concentração: Programação.

Americana, 06 de dezembro de 2021.

Banca Examinadora:

____
Antonio Alfredo Lacerda
Especialista
Faculdade de Tecnologia de Americana “Ministro Ralph Biasi”

____
Clerivaldo Jose Roccia
Mestre
Faculdade de Tecnologia de Americana “Ministro Ralph Biasi”

____
Maria Elizete Luz Saes

Mestre

Faculdade de Tecnologia de Americana “Ministro Ralph Biasi”

RESUMO

Com o avanço da tecnologia, foram surgindo diversas ferramentas que podem otimizar o tempo de desenvolvimento de aplicações, definir novos padrões e aperfeiçoar códigos com testes. Uma dessas ferramentas é o *Framework* Laravel, sendo muito utilizado para aplicações *web*, tais como *E-commerces*, sites pessoais e sistemas como serviços. O presente trabalho de conclusão de curso refere-se ao estudo e desenvolvimento de uma Aplicação simples utilizando o *Framework* Laravel, com foco nos principais materiais de desenvolvimento e conceitos do próprio *Framework*. Apesar do Laravel possuir uma documentação completa, é necessário que sejam estudados outros *Softwares*, linguagens de programação e conceitos que contribuem para a elaboração de uma aplicação, independente de qual ferramenta seja utilizada. Nesse sentido o estudo inicia com uma revisão de literatura do acervo de documentos disponíveis em bibliotecas virtuais e na plataforma Google Acadêmico acerca da linguagem de programação utilizada, que é o PHP, e os conceitos e padrões utilizados no Laravel. Com o objetivo de desenvolver uma aplicação funcional, foram descritos os principais *Softwares* necessários para seu desenvolvimento como Composer, Docker, Laravel Sail e MySQL. Além disso, foram abordados conceitos que envolvem a Arquitetura de Software MVC, *Facades*, Rotas, Middlewares e Validações de Requisições. Por fim, foi apresentada uma aplicação de blog hipotética, para melhor compreensão de conceitos, onde é possível registrar, realizar o acesso, gerenciar e visualizar todas as postagens. É esperado que o aprendizado adquirido no desenvolvimento da Aplicação, seja utilizado para futuros Projetos, deixando-os cada vez mais aprimorados, atualizados, seguros, com alto desempenho e de desenvolvimento ágil.

Palavras Chave: Framework; Laravel; Arquitetura de Software MVC;

ABSTRACT

With the advancement of technology, several tools have emerged that can optimize application development time, define new standards and improve code with tests. One of these tools is the Laravel Framework, which is widely used for web applications, such as E-commerces, personal websites and systems as services. This end of course work refers to the study and development of a simple Application using the Laravel Framework, focusing on the main development materials and concepts of the Framework itself. Although Laravel has a complete documentation, it is necessary to study other software, programming languages and concepts that contribute to the development of an application, regardless of which tool is used. In this sense, the study begins with a literature review of the collection of documents available in virtual libraries and on the Academic Google platform about the programming language used, which is PHP, and the concepts and patterns used in Laravel. In order to develop a functional application, the main Softwares needed for its development were described, such as Composer, Docker, Laravel Sail and MySQL. In addition, concepts involving the MVC Software Architecture, Facades, Routes, Middlewares and Validations of Requests. Finally, a hypothetical blog application was presented, for a better understanding of concepts, where it is possible to register, access, manage and view all posts. It is expected that the learning acquired in the development of the Application will be used for future Projects, making them increasingly improved, updated, safe, with high performance and agile development.

Keywords: Framework; Laravel; MVC Software Architecture;

LISTA DE ILUSTRAÇÕES

Figura 1 - Exemplo de código PHP no HTML........14
Figura 2 - Logotipo do *Framework* Laravel........15
Figura 3 - Código utilizada no terminal para instalação........17
Figura 4 - Exemplo injeção de dependência........18
Figura 5 - Exemplo do registro de um *Singleton*........19
Figura 6 - Criação de uma URI........20
Figura 7 - URI com parâmetro........21
Figura 8 - Comando para criar uma *migration*........22
Figura 9 - Exemplo *migration* Laravel........23
Figura 10 - Utilização do *Facade* DB........24
Figura 11 - Comando para gerar uma *model*........25
Figura 12 - Exemplo da utilização do ORM........25
Figura 13 - Exemplo código de uma *View*........26
Figura 14 - Exemplo rota chamando *View*........26
Figura 15 - Exemplo diretiva *if* no *blade*........27
Figura 16 - Rota chamando controladora........27
Figura 17 - *Controller* retornando uma *View*........28
Figura 18 - Exemplo de uma *middleware*........29
Figura 19 - Comando para criar uma *middleware*........29
Figura 20 - Pegando o valor "*name*" de uma *request*........30
Figura 21 - Chamando *helper old* em um *input*........30
Figura 22 - Adquirindo arquivo em uma *request*........31
Figura 23 - Exemplo de utilização de validações........31
Figura 24 - Exemplo de exibição dos erros em uma *view*.........32
Figura 25 - Relacionamento 1-N........33
Figura 26 - Exemplo da utilização do método *with*........33
Figura 27 - *Queries* executadas com o método *with*........34
Figura 28 - Comando para inserir a dependência Laravel Breeze........34
Figura 29 - Comandos para instalar o Laravel Breeze........35
Figura 30 - Comando para a criação da *migration*........35
Figura 31 - *Migration* da tabela *posts*........36
Figura 32 - Comando para executar as *migrations*........37
Figura 33 - Comando para criar uma *Model*........37

Figura 34 - Definição da *Model Post* 37
Figura 35 - Metódo *posts* na *model User* 38
Figura 36 - Comandos para a criação dos *Controllers* 38
Figura 37 - Classe *HomeController* 39
Figura 38 - Método *PostController::show* 40
Figura 39 - Método *PostController::create* 40
Figura 40 - Método *PostController::post* 41
Figura 41 - Método *PostController::edit* 42
Figura 42 - Método *PostController::update* 42
Figura 43 - Método *PostController::destroy* 43
Figura 44 - *View home.index* 44
Figura 45 - *View post.index* 45
Figura 46 - *View post.create* 46
Figura 47 - *View post.update* 47
Figura 48 - Rotas dos *posts* 47
Figura 49 - Página sem nenhuma postagem 48
Figura 50 - Página de Registro 49
Figura 51 - Página de todos os *posts* do usuário logado 49
Figura 52 - Página de cadastro de postagem 50
Figura 53 - Página de cadastro de postagem com erro 50
Figura 54 - Criação de uma postagem com sucesso 51
Figura 55 - *Home* com postagens 51
Figura 56 - Imagem interna da postagem 52

LISTA DE ABREVIATURAS E SIGLAS

PHP – PHP: *Hypertext Preprocessor*

HTML – *HyperText Markup Language*

API – *Application Programming Interface*

URI – *Uniform Resource Identifier*

CLI – *Command-line Interface*

OOP – *Object Oriented Programming*

SO – Sistema Operacional

MVC – *Model, View, Controller*

SGBD – Servidor e Gerenciador de Banco de Dados

HTTP – *HyperText Transfer Protocol*

CSRF – *Cross-site Request Forgery*

ORM – *Object-Relational Mapper*

SQL – *Structure Query Language*

CRUD – *Create, Read, Update* e *Delete*

SUMÁRIO

1 INTRODUÇÃO 12

2 CONHECENDO PHP E FRAMEWORK 13

2.1 A LINGUAGEM DE PROGRAMAÇÃO PHP 13

2.2 CARACTERÍSTICAS DE UM FRAMEWORK 14

3 LARAVEL E FERRAMENTAS PARA O DESENVOLVIMENTO 15

3.1 COMPOSER 15

3.2 DOCKER E LARAVEL SAIL 16

4 EXPLORANDO O LARAVEL 17

4.1 CICLO DE UMA REQUISIÇÃO 17

4.2 SERVICE CONTAINER 18

4.3 SERVICE PROVIDER 18

4.4 FACADE 20

4.5 ROTAS 20

4.6 BANCO DE DADOS 21

4.6.1 MySQL 21

4.6.2 Migrations 21

4.6.3 Facade DB 24

4.7 ARQUITETURA DE SOFTWARE MVC 24

4.7.1 Model 25

4.7.2 View 26

4.7.3 Controller 27

4.8 MIDDLEWARE 28

4.9 REQUEST 30

4.10 VALIDAÇÕES 31

4.11 RELACIONAMENTOS ENTRE MODELOS 32

4.12 LARAVEL NA PRÁTICA 34

4.12.1 Sistema de Login ... 34
4.12.2 Criação da Migration ... 35
4.12.3 Criação das Model ... 37
4.12.4 Criação dos Controllers ... 38
4.12.4.1 HomeController ... 39
4.12.4.2 PostController ... 39
4.12.5 Criação das Views ... 43
4.12.6 Rotas ... 47
4.12.7 Projeto Final e Telas ... 48
5 CONCLUSÃO ... 53
REFERÊNCIAS BIBLIOGRÁFICAS ... 54

1 INTRODUÇÃO

Com o aumento da demanda de aplicações web, o mercado de trabalho tornou-se cada vez mais desafiador para os desenvolvedores. Sendo assim, surgiram ferramentas que contribuem para a automatização e padronização de projetos, minimizando os erros, aprimorando a gestão de tempo e a facilidade para outros desenvolvedores modificarem o código. Uma dessas ferramentas é o uso de *Frameworks*.

Um *Framework* facilita o desenvolvimento de aplicações, com a reutilização de classes, métodos e funções, definindo um tipo de Design de Projeto para padronização. O Laravel é um *Framework* robusto que fornece diversas ferramentas, pacotes e estrutura de códigos prontas para facilitar o desenvolvimento de Projetos.

Mas para que seja necessário a criação de um Projeto em Laravel, é necessário ter o conhecimento de como ele funciona internamente, estruturas de Banco de Dados existentes, o funcionamento de sua estrutura Modelo, Visão e Controladora, entre outros conceitos essenciais.

Dessa forma, ao adquirir o conhecimento básico do Laravel, um sistema pode ser desenvolvido rapidamente e de forma segura. Com o surgimento de novas demandas para o desenvolvimento de Projetos, as implementações tornam-se mais simples devido à organização de estrutura de códigos.

Sendo assim, serão abordados os seguintes temas, no Capítulo 1 – Introdução, Capítulo 2 – Conhecendo PHP e *Framework*, Capítulo 3 – Ferramentas para o Desenvolvimento de um Projeto em Laravel, Capítulo 4 – Estrutura de um Projeto e o Desenvolvimento de um sistema simples hipotético para exemplificação e Capítulo 5 – Conclusão.

2 CONHECENDO PHP E FRAMEWORK

O PHP (PHP: *Hypertext Preprocessor*) é uma linguagem de programação voltada para o desenvolvimento web, sendo código aberto e pode ser embutida dentro do HTML (*HyperText Markup Language*). Essa linguagem é simples de ser utilizada e amigável para novos programadores (LERDORF et al., 2002).

Um *Framework* está presente em diversas linguagens de programação, não sendo diferente em PHP. Seu principal objetivo está relacionado a facilidade do desenvolvimento e a reutilização de classes, métodos e funções, normalmente um *Framework* define um tipo de design de Projeto para padronização (GABARDO, 2017).

2.1 A LINGUAGEM DE PROGRAMAÇÃO PHP

Segundo Lerdorf (2002), o PHP é utilizado no *server-side*, isso significa que todo o código é executado na parte do servidor, fazendo com que toda e qualquer responsabilidade da lógica de manipulação de dados sensíveis fique focada apenas no *back-end*.

Além do seu foco no desenvolvimento web, é possível desenvolver códigos para serem executados em CLI (*Command-line Interface*) e também para desenvolvimento de aplicações *Desktop* a partir de extensões criadas pela comunidade, tais como a PHP-GTK (“PHP: O que o PHP pode fazer - Manual”, [s.d.]).

A escrita do código em PHP pode ser feita tanto em estrutura procedural ou em OOP (*Object Oriented Programming*). Pensando na orientação a objeto, ela não é uma linguagem com tipagem forte, isso significa que é possível criar uma variável com o seu tipo alterado constantemente, cabe ao desenvolvedor definir o tipo da variável, tornando-a à tipagem obrigatória (“PHP: Introdução - Manual”, [s.d.]). Além do mais, um outro ponto positivo é seu suporte a uma variedade de Banco de Dados e a possibilidade de comunicação por vários protocolos (“PHP: O que é o PHP - Manual”, [s.d.]).

Um exemplo de código de PHP em meio do HTML pode ser visto na Figura 1:

Figura 1 - Exemplo de código PHP no HTML

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>Exemplo</title>
    </head>
    <body>

        <?php
            echo "Olá, eu sou um script PHP!";
        ?>

    </body>
</html>
```

Fonte: (“PHP: O que é o PHP - Manual”, [s.d.])

2.2 CARACTERÍSTICAS DE UM FRAMEWORK

Segundo Gabardo (2017), um *Framework* fornece códigos que podem ser reutilizados em outros Projetos e alguns estão relacionados com validações, criações de formulários e autenticações.

Existem alguns *Frameworks* utilizados frequentemente para o desenvolvimento web em PHP, tais como Laravel, Symfony, Laminas, Zend, CakePHP, Yii, entre outros. Cada um tem suas vantagens, desvantagens e padrões sendo que a equipe desenvolvedora decidirá qual o *Framework* ideal para o respectivo Projeto (“Conheça 9 Frameworks para fazer site em PHP”, 2020).

3 LARAVEL E FERRAMENTAS PARA O DESENVOLVIMENTO

Projeto *open-source*, criado em 2011 por Taylor B. Otwell, o Laravel possui o padrão de estrutura de código em MVC (*Model*, *View* e *Controller*), focado em performance, escalabilidade e segurança. Um dos benefícios do Laravel é ser modular, ou seja, um fragmento de código pode ser utilizado em outros Projetos por meio de pacotes (“Laravel”, 2020).

Constantemente o Laravel recebe atualizações de segurança e por ser um Projeto de código aberto, qualquer desenvolvedor pode contribuir. O Laravel é muito amigável para novatos, tendo conceitos simples, intuitivos e uma documentação objetiva. No entanto, ele também pode ser desafiador para programadores mais experientes com conceitos avançados, como injeção de dependência, testes unitários, filas e eventos (“Installation - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Figura 2 - Logotipo do *Framework* Laravel

Fonte: (“Installation - Laravel - The PHP Framework For Web Artisans”, [s.d.])

Assim como outros *Frameworks*, o Laravel possui suporte para inúmeras ferramentas de desenvolvimento, tais serviços estão relacionados com Banco de Dados, serviço de cache e testes, sendo necessário configurá-los logo após a criação de um Projeto (“Installation - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Para que os arquivos configurados façam sentido posteriormente, é necessário ter um aprofundamento sobre tais serviços, podendo assim ter mais facilidade em manter os códigos padronizados, organizados, testáveis e suportado em SOs (Sistemas Operacionais) diversificados (“Installation - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

3.1 COMPOSER

Essa ferramenta é utilizada para o gerenciamento de dependências de Projetos, que organiza quais bibliotecas ou pacotes devem ser instalados e/ou

atualizados. Diferente de outros gerenciadores de pacotes como o Yum ou Apt, ele é focado no gerenciamento de pacotes **por Projeto**, gerando um diretório dentro de um Projeto específico e deixando todas as dependências dentro dele (“Introduction - Composer”, [s.d.]).

3.2 DOCKER E LARAVEL SAIL

O Docker serve para gerenciar a infraestrutura de uma aplicação, de forma que não tenha uma grande perda de desempenho na máquina e que ao mesmo tempo consiga manter o gerenciamento de versões dos aplicativos sem conflitos (“O que é Docker · Docker para Desenvolvedores”, [s.d.]).

Um *container* do Docker, é um pacote de software leve, que inclui o que for necessário para a execução do aplicativo, como código, ferramentas do sistema, bibliotecas e configurações (“What is a Container?”, [s.d.]). Cada *container* utiliza o *kernel* do Linux como Cgroups e *namespaces* para executar de maneira independente processos no sistema, mantendo os Projetos executados simultaneamente (“O que é Docker”, [s.d.]).

O Laravel Sail é uma interface de linha de comando que facilita a interação com o ambiente de desenvolvimento de uma aplicação feita em Laravel com o Docker. Ao utilizar o Laravel Sail, as seguintes aplicações já vem configuradas: PHP, MySQL, Redis (“Laravel Sail - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

4 EXPLORANDO O LARAVEL

Antes de iniciar qualquer tipo de desenvolvimento de uma aplicação em Laravel, é necessário realizar a instalação do mesmo. O Laravel fornece diversas maneiras de instalação, a mais recente sendo a instalação de um container do Docker com sua própria interface de linha de comando (“Installation - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Para realizar a instalação por esse método, apenas é preciso ter o Docker na máquina de desenvolvimento e digitar em um terminal, no diretório que o Projeto será criado o seguinte comando:

Figura 3 - Código utilizada no terminal para instalação

```
curl -s https://laravel.build/example-app | bash
```

Fonte: (“Installation - Laravel - The PHP Framework For Web Artisans”, [s.d.])

Com esse comando executado em um terminal, ele vai criar um diretório na pasta onde o terminal está sendo executado com o nome de “*example-app*”, e dentro dele tem todos os arquivos necessários para a criação de uma aplicação Laravel. E por ser um container, podemos ter várias aplicações sendo executadas ao mesmo tempo.

Após a conclusão do comando, entrando no diretório do Projeto e executando “*./vendor/bin/sail up*”, será iniciado o container do Laravel e a aplicação será acessada na porta configurada no Docker, seu padrão é a porta 80.

4.1 CICLO DE UMA REQUISIÇÃO

Para que fique mais fácil o desenvolvimento de uma aplicação no Laravel, é importante identificar o ciclo de uma requisição.

Tudo se inicia com uma requisição HTTP (*HyperText Transfer Protocol*), que é enviada para o arquivo “*public/index.php*”, esse arquivo cria uma instância da aplicação, conhecida também como *Service Container*. Logo em seguida a requisição é enviada para o núcleo da instância, localizada em “*app/Http/kernel.php*”, essa classe inicializa as configurações de erros, logs, detecta em qual ambiente a aplicação está sendo executada. Além disso, o núcleo da aplicação define quais são as *middlewares*

que aquela requisição irá passar (“Request Lifecycle - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Uma vez que a rota ou controlador retorna uma resposta, essa resposta passa pelos *middlewares* e uma vez aprovado por todos, o arquivo “index.php” envia a resposta para o usuário (“Request Lifecycle - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

4.2 SERVICE CONTAINER

Para a injeção de dependência no Laravel , é utilizado o *service container*[1], uma vez colocado no construtor do código qual a classe que ela deve receber, automaticamente o Laravel irá instanciar e passar como parâmetro (“Service Container - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Figura 4 - Exemplo injeção de dependência

```
public function __construct(UserRepository $users)
{
    $this->users = $users;
}
```

Fonte: (“Service Container - Laravel - The PHP Framework For Web Artisans”, [s.d.])

4.3 SERVICE PROVIDER

O *service provider* serve para inicializar todo o Laravel, sendo que tudo que precisa ser executado ou registrado durante a instância da aplicação é feito nos *service provider* (“Service Providers - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Pode ser criado um *service provider* para iniciar coisas no Laravel, sem depender necessariamente do Laravel, podendo assim criar pacotes que podem ser reutilizados em outros Projetos (“Como criar um pacote para o Laravel”, 2019).

Dentro do *service provider* é feito o *bind* para a injeção de dependência, *singletons* e realizamos o *boot* de rotas, *listeners* ou qualquer outra peça de

[1] Ferramenta utilizada para injeção de dependência dentro do código do Laravel.

funcionalidade do sistema (“Service Providers - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Figura 5 - Exemplo do registro de um *Singleton*

```php
<?php

namespace App\Providers;

use App\Services\Riak\Connection;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton(Connection::class, function ($app) {
            return new Connection(config('riak'));
        });
    }
}
```

Fonte: (“Service Providers - Laravel - The PHP Framework For Web Artisans”, [s.d.])

4.4 FACADE

Fachada ou *Facade* é um *design pattern* cujo papel é fazer uma biblioteca, pacote ou *Framework* se resumir em apenas interface e/ou classe, para que seja possível o sistema reconhecer apenas o necessário para realizar as ações e não sobre as regras de negócio da biblioteca (“Facade”, [s.d.]).

O Laravel fornece uma variedade de classes *Facades*, App, Auth, Cache, DB, Redirect, Response, Route, entre outras que podem ser utilizadas em qualquer parte do código, cada um tem um propósito específico, mas uma não interage diretamente com a outra e a aplicação não sabe e não interfere nas regras de cada *Facade* (“Facades - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Além das *Facade*, o Laravel fornece *helpers* para que seja possível chamar as *Facade* sem precisar da instância da classe, fazendo com que o código fique mais limpo (“Facades - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

4.5 ROTAS

As rotas são definidas em um arquivo onde se passa a URI (*Uniform Resource Identifier*) e uma *closure* ou um controlador com o método, o arquivo padrão de rotas do Laravel é “routes/web.php” e automaticamente as rotas passam pelas *middlewares* do tipo *web*, possuindo proteção CSRF (*Cross-site Request Forgery*) e os dados da sessão (“Routing - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Figura 6 - Criação de uma URI

```php
use Illuminate\Support\Facades\Route;

Route::get('/greeting', function () {
    return 'Hello World';
});
```

Fonte: (“Routing - Laravel - The PHP Framework For Web Artisans”, [s.d.])

Para passar parâmetros em uma URI, é necessário definir no parâmetro de definição da rota o nome da variável com chaves, assim os parâmetros passados na URL serão interpretados pelos controladores ou pela *closure*.

Figura 7 - URI com parâmetro

```
use Illuminate\Http\Request;

Route::get('/user/{id}', function (Request $request, $id) {
    return 'User '.$id;
});
```

Fonte: (“Routing - Laravel - The PHP Framework For Web Artisans”, [s.d.])

4.6 BANCO DE DADOS

O suporte para gerenciamento de Banco de Dados para o Laravel é amplo, tendo suporte para MySQL, PostgreSQL, SQLite e SQLServer nativamente (“Database: Getting Started - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Mas o seu suporte não se limita apenas a esses Bancos de Dados, pois a comunidade cria pacotes que adiciona suporte para outros Bancos de Dados, como o mongoDB (SEGERS, 2021).

4.6.1 MySQL

O MySQL é um SGBD (Servidor e Gerenciador de Banco de Dados) relacional de licença dupla, utilizado por empresas de diversos portes (MILANI, 2007). Segundo Milani (2007), o MySQL é o Banco de Dados *open-source* mais utilizado pela internet, para soluções *web* e *e-commerce*, por ser um gerenciador de Banco de Dados de rápida resposta.

O suporte para as plataformas para esse gerenciador de Banco de Dados é muito amplo, possuindo flexibilidade para diversos SOs, tais como Ubuntu, Apple macOS e Microsoft Windows. Além disso, ele fornece uma API (*Application Programming Interface*) para conectar com várias linguagens de programação, como C#, PHP, Java e Python (MEHTA et al., 2018).

4.6.2 Migrations

Uma *migration* assemelha-se a um controlador de versão para a estrutura do Banco de Dados, com a possibilidade de compartilhar o Projeto entre as equipes e executando um comando para que seja possível manter todos com a mesma estrutura

de Banco de Dados (“Database: Migrations - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Para a criação de uma *migration*, é necessário executar o comando:

Figura 8 - Comando para criar uma *migration*

```
php artisan make:migration create_flights_table
```

Fonte: (“Database: Migrations - Laravel - The PHP Framework For Web Artisans”, [s.d.])

A estrutura do código da *migration* para a criação da estrutura do Banco de Dados é a seguinte:

Figura 9 - Exemplo *migration* Laravel

```php
<?php

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

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}
```

Fonte: (“Database: Migrations - Laravel - The PHP Framework For Web Artisans”, [s.d.])

Quando executada essa *migration*, será criado uma tabela no banco, contendo 5 campos, sendo:

- *unsigned integer id not null autoincrement*
- *string* 100 *name not null*
- *string* 100 *airline not null*
- *timestamp created_at not null*
- *timestamp updated_at not null*

4.6.3 Facade DB

A *Facade* DB pode ser utilizada para realizar *queries* SQL (*Structure Query Language*) para o gerenciador de Banco de Dados, essa chamada para o Banco de Dados é feita de forma segura, utilizando o SQL *Bind* para evitar falhas de segurança, tais como SQL *Injection* ("Database: Query Builder - Laravel - The PHP Framework For Web Artisans", [s.d.]).

Todos os tipos de *queries* SQL podem ser executados, *select, create, update, delete*, podendo também executar queries com *database transaction* e assim realizar *rollback* da transação com o Banco de Dados, caso algo de errado aconteça durante o processo ("Database: Query Builder - Laravel - The PHP Framework For Web Artisans", [s.d.]).

Figura 10 - Utilização do *Facade* DB

```
public function index()
{
    $users = DB::table('users')->get();

    return view('user.index', ['users' => $users]);
}
```

Fonte: ("Database: Query Builder - Laravel - The PHP Framework For Web Artisans", [s.d.])

4.7 ARQUITETURA DE SOFTWARE MVC

As siglas MVC significam *Model*, *View* e *Controller* respectivamente, onde a *Model* ou Modelo é a camada responsável pelas regras de negócios, normalmente um Modelo está diretamente relacionado com uma tabela de Banco de Dados. O *Controller* ou controladora é responsável pela comunicação da rota com a *View*,

dentro dos métodos da Controladora é utilizado os Modelos. E as *Views* ou Visões é a camada responsável para a renderização do que o usuário final irá visualizar ("O que é padrão MVC?", [s.d.]).

4.7.1 Model

Para o Laravel, a *Model* é diretamente chamado de Eloquente ORM (*Object-Relational Mapper*), ou seja, um Mapeamento Objeto-Relacional, isso significa que cada Model tem uma tabela no Banco de Dados ("Eloquent: Getting Started - Laravel - The PHP Framework For Web Artisans", [s.d.]).

Uma *Model* pode ser gerada por linha de comando, o comando abaixo exemplifica a criação de uma chamada *Flight*:

Figura 11 - Comando para gerar uma *model*

```
php artisan make:model Flight
```

Fonte: ("Eloquent: Getting Started - Laravel - The PHP Framework For Web Artisans", [s.d.])

Como a *Model* está diretamente associada com uma tabela no Banco de Dados, podemos realizar as *queries* diretamente por ela:

Figura 12 - Exemplo da utilização do ORM

```
$flights = Flight::where('active', 1)
               ->orderBy('name')
               ->take(10)
               ->get();
```

Fonte: ("Eloquent: Getting Started - Laravel - The PHP Framework For Web Artisans", [s.d.])

A execução do código acima vai trazer do Banco de Dados, dez voos ativos, ordenados por nome.

4.7.2 View

Para organização, o Laravel armazena os arquivos de Visão no diretório "*resources/views*", onde os arquivos *views* são reconhecidos pelo formato "*.blade.php*", esses arquivos servem para retornar a página para o usuário final, entretanto podendo ter alterações e lógicas ("Views - Laravel - The PHP Framework For Web Artisans", [s.d.]).

Figura 13 - Exemplo código de uma *View*

```
<!-- View stored in resources/views/greeting.blade.php -->

<html>
    <body>
        <h1>Hello, {{ $name }}</h1>
    </body>
</html>
```

Fonte: ("Views - Laravel - The PHP Framework For Web Artisans", [s.d.])

Na rota ou na controladora, é chamado o *helper*, passando como parâmetro a *view* que será renderizada, junto com os parâmetros necessários para renderizar a mesma ("Views - Laravel - The PHP Framework For Web Artisans", [s.d.]).

Figura 14 - Exemplo rota chamando *View*

```
Route::get('/', function () {
    return view('greeting', ['name' => 'James']);
});
```

Fonte: ("Views - Laravel - The PHP Framework For Web Artisans", [s.d.])

O *blade* é o *template engine*, também conhecido por motor de modelagem, e ele tem sua sintaxe própria, facilitando o desenvolvimento de uma página devido a sua fácil e intuitiva diretivas (“Blade Templates - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Figura 15 - Exemplo diretiva *if* no *blade*

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

Fonte: (“Blade Templates - Laravel - The PHP Framework For Web Artisans”, [s.d.])

4.7.3 Controller

O *controller* é utilizado para que as regras de negócio não fiquem definidas nas rotas diretamente, sendo assim as rotas ficam responsáveis apenas de saber qual URI deve chamar qual controlador (“Controllers - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Figura 16 - Rota chamando controladora

```
use App\Http\Controllers\UserController;

Route::get('/user/{id}', [UserController::class, 'show']);
```

Fonte: (“Controllers - Laravel - The PHP Framework For Web Artisans”, [s.d.])

Como toda a regra de negócio fica na controladora do Laravel, então a exibição, criação, atualização, remoção entre outras ações de algum modelo fica responsável exclusivamente do Controlador vinculado (BEAN, 2015).

Figura 17 - *Controller* retornando uma *View*

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     *
     * @param  int  $id
     * @return \Illuminate\View\View
     */
    public function show($id)
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

Fonte: (“Controllers - Laravel - The PHP Framework For Web Artisans”, [s.d.])

4.8 MIDDLEWARE

A *Middleware* é utilizada para filtrar as requisições HTTP, fazendo com que todas as chamadas HTTP passem por especificas regras antes de chegar as regras de negócio das controladoras (“Middleware no Laravel (Filtros)”, 2017).

O Laravel fornece inúmeras *middlewares* relacionadas a autenticação, sessão e proteção CSRF e estão localizadas no diretório “*app/Http/Middleware*” (“Middleware - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Figura 18 - Exemplo de uma *middleware*

```php
<?php

namespace App\Http\Middleware;

use Closure;

class EnsureTokenIsValid
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->input('token') !== 'my-secret-token') {
            return redirect('home');
        }

        return $next($request);
    }
}
```

Fonte: (“Middleware - Laravel - The PHP Framework For Web Artisans”, [s.d.])

A base de uma *middleware* pode ser criada a partir do comando “*make:middleware*”, seguido do nome.

Figura 19 - Comando para criar uma *middleware*

```
php artisan make:middleware EnsureTokenIsValid
```

Fonte: (“Middleware - Laravel - The PHP Framework For Web Artisans”, [s.d.])

4.9 REQUEST

A classe *Request* do Laravel é responsável pela interação da requisição HTTP atual, podendo ter controle de *header, inputs*, *cookies*, arquivos e outras informações sobre a *request* que foi solicitada. Toda *Request* é inserido via injeção de dependência nas *closure* de uma rota ou no método de um controlador (“HTTP Requests - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Figura 20 - Pegando o valor "*name*" de uma *request*

```
public function store(Request $request)
{
    $name = $request->input('name');

    //
}
```

Fonte: (“HTTP Requests - Laravel - The PHP Framework For Web Artisans”, [s.d.])

Para ter acesso aos valores anteriores de uma requisição, o Laravel fornece o *helper “old”*, onde com ele, é passado um parâmetro contendo qual o campo do valor enviado e então ele retorna seu valor antigo (“HTTP Requests - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Figura 21 - Chamando *helper old* em um *input*

```
<input type="text" name="username" value="{{ old('username') }}">
```

Fonte: (“HTTP Requests - Laravel - The PHP Framework For Web Artisans”, [s.d.])

Para interagir com arquivos, basta apenas chamar o método *file* da *request* ou então utilizar o conceito de método mágico do PHP definido na *Model* do Laravel (“HTTP Requests - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Figura 22 - Adquirindo arquivo em uma *request*

```
$file = $request->file('photo');

$file = $request->photo;
```

Fonte: (“HTTP Requests - Laravel - The PHP Framework For Web Artisans”, [s.d.])

4.10 VALIDAÇÕES

Existem diferentes maneiras de validar os valores que chegam de uma *request*, a mais comum no Laravel é utilizando a própria requisição para isso, sendo assim, o código fica mais limpo, organizado e divide as responsabilidades (“Validation - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Além de ser possível criar regras de validações de regra de negócio nas controladoras, o Laravel fornece inúmeras já definidas e documentadas, algumas delas são *required, unique,* e *max*, que validam respectivamente o valor que é obrigatório, e deve ser único, um limite de tamanho caso numérico de caracteres caso seja texto (“Validation - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Figura 23 - Exemplo de utilização de validações

```
$validatedData = $request->validate([
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

Fonte: (“Validation - Laravel - The PHP Framework For Web Artisans”, [s.d.])

Caso seja encontrado algum erro durante uma requisição, o Laravel automaticamente faz um redirecionamento para a página de origem da requisição, contendo os valores antigos e uma lista de erros em uma variável definida com o nome de “*$error*”.

Figura 24 - Exemplo de exibição dos erros em uma *view*.

```
<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->
```

Fonte: (“Validation - Laravel - The PHP Framework For Web Artisans”, [s.d.])

4.11 RELACIONAMENTOS ENTRE MODELOS

Um Banco de Dados relacional normalmente terá tabelas que se relacionam entre si, por exemplo uma tabela de usuários com *posts* de um *blog* (“Eloquent: Relationships - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Para que esse tipo de relacionamento funcione entre os modelos do Laravel, simplesmente precisa apenas criar um método e então passar no retorno do método o tipo de relacionamento.

Figura 25 - Relacionamento 1-N

```
class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

Fonte: (“Eloquent: Relationships - Laravel - The PHP Framework For Web Artisans”, [s.d.])

No exemplo acima, está sendo criado o relacionamento entre os modelos, onde um *post* tem vários (pelo método em inglês *hasMany*) comentários (“Eloquent: Relationships - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Para que seja possível evitar o problema de N+1, é necessário fazer o *Eager Loading*. O problema N+1 pode ser ilustrado da seguinte forma: existem 25 livros no Banco de Dados, cada livro tem um autor, se realizar uma busca de todos os livros, e para cada livro realizar a busca de seu autor, será realizado 26 *queries* totais, ou seja, N+1 (“Eloquent: Relationships - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Para evitar esse problema, simplesmente precisamos chamar o método “*with”*, com isso o Laravel irá realizar apenas 2 buscas no Banco de Dados.

Figura 26 - Exemplo da utilização do método *with*

```
$books = Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}
```

Fonte: (“Eloquent: Relationships - Laravel - The PHP Framework For Web Artisans”, [s.d.])

Figura 27 - *Queries* executadas com o método *with*

```
select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)
```

Fonte: (“Eloquent: Relationships - Laravel - The PHP Framework For Web Artisans”, [s.d.])

4.12 LARAVEL NA PRÁTICA

Para a exemplificação, será criado uma aplicação de *blog* hipotética com uma página de *Login*, um sistema de cadastro de *Post*, e uma página onde pode visualizar todos os *posts.*

Cada postagem terá uma imagem, título, descrição, autor e as informações de criação e atualização.

4.12.1 Sistema de Login

Para o sistema de Login, será utilizado um pacote para o desenvolvimento inicial do Laravel, chamado de Laravel Breeze. Esse pacote fornece um sistema de autenticação com a página de dashboard e tudo customizado com TailwindCSS[2] (“Starter Kits - Laravel - The PHP Framework For Web Artisans”, [s.d.]).

Após a criação de um Projeto Laravel deve ser executado o seguinte comando:

Figura 28 - Comando para inserir a dependência Laravel Breeze

```
composer require laravel/breeze --dev
```

Fonte: (“Starter Kits - Laravel - The PHP Framework For Web Artisans”, [s.d.])

Após a conclusão da inserção do pacote Laravel Breeze como dependência do Projeto, é necessário instalar ele e executar a compilação dos pacotes *front-end.*

---

[2] Framework CSS

Figura 29 - Comandos para instalar o Laravel Breeze

```
php artisan breeze:install

npm install
npm run dev
php artisan migrate
```

Fonte: (“Starter Kits - Laravel - The PHP Framework For Web Artisans”, [s.d.])

Após a conclusão da instalação, as rotas “/*register*” e “/*login*” estarão disponíveis para registrar e realizar o login.

4.12.2 Criação da Migration

Será preciso a *Migration* para criar a tabela no Banco de Dados, como o nome da tabela será *posts*, podemos executar o comando:

Figura 30 - Comando para a criação da *migration*

```
sail@c18fe248fbd4:/var/www/html$ php artisan make:migration create_posts_table
```

Fonte: Próprio Autor

Esse comando irá gerar o arquivo de criação da tabela *posts*, mas será necessário definir as colunas e os tipos delas.

Figura 31 - *Migration* da tabela *posts*

```
class CreatePostsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->foreignId('user_id')->constrained()->onUpdate('CASCADE')->onDelete('CASCADE');
            $table->string('image')->nullable();
            $table->string('title');
            $table->longText('description');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('posts');
    }
}
```

Fonte: Próprio Autor

A *migration* acima define que a tabela *posts* terá as colunas:

- *unsigned integer id not null autoincrement*

- *foreign key unsigned integer user_id not null reference users.id on delete cascade on update cascade*

- *string* 100 *image null*

- *string* 100 *title not null*

- *longtext description not null*

- *timestamp created_at not null*

- *timestamp updated_at not null*

Para que o PHP execute as *migrations* e então crie as tabelas no Banco de Dados, é necessário executar o comando abaixo:

Figura 32 - Comando para executar as *migrations*

```
sail@c18fe248fbd4:/var/www/html$ php artisan migrate
```

Fonte: Próprio Autor

4.12.3 Criação das Model

Após a criação da *Migration*, é necessário criar *Model* que representa essa tabela, para isso é executado o comando:

Figura 33 - Comando para criar uma *Model*

```
sail@c18fe248fbd4:/var/www/html$ php artisan make:model Post
```

Fonte: Próprio Autor

E então configuramos a *model* como definido abaixo:

Figura 34 - Definição da *Model Post*

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    protected $guarded = [];

    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Fonte: Próprio Autor

Como descrito acima, *Post* pertence a um Usuário, ou seja, seu autor, mas é preciso definir também que um Usuário tem vários *Posts*, para isso criamos o método a seguir em Usuário:

Figura 35 - Metódo *posts* na *model User*

```
public function posts()
{
    return $this->hasMany(Post::class);
}
```

Fonte: Próprio Autor

4.12.4 Criação dos Controllers

Para esse Projeto, será criado duas controladoras, sendo que uma será responsável para a visualização de todos as postagens, podendo se dizer público, pois não é necessário estar logado para ter acesso as páginas, essa controladora será chamada de *HomeController.*

A segunda controladora chamada de *PostController,* será responsável por realizar as ações privadas de um *post*, sendo assim, será necessário estar autenticado para ter acesso as ações, e assim será possível criar, atualizar, deletar posts específicos, além disso, ele terá um método público específico para a visualização de uma postagem.

Para a criação desses *Controllers* é utilizado os comandos:

Figura 36 - Comandos para a criação dos *Controllers*

```
sail@c18fe248fbd4:/var/www/html$ php artisan make:controller HomeController
sail@c18fe248fbd4:/var/www/html$ php artisan make:controller PostController
```

Fonte: Próprio Autor

4.12.4.1 HomeController

A controladora *HomeController* será simples, tendo apenas um método responsável para exibir todos os *posts* encontrado no Banco de Dados.

Figura 37 - Classe *HomeController*

```php
<?php

namespace App\Http\Controllers;

use App\Models\Post;

class HomeController extends Controller
{
    public function index()
    {
        $posts = Post::with('user')->orderByDesc('created_at')->get();
        return view('home.index', compact('posts'));
    }
}
```

Fonte: Próprio Autor

O método *index* está buscando todos os posts encontrado no Banco de Dados e cada post tem um usuário, por isso a utilização do método *with*, assim será evitado o problema de N+1 e as postagens serão ordenadas por data da criação.

Logo em seguida é chamado a *view* “*home.index*” e todos os *posts* são enviados para ela.

4.12.4.2 PostController

A controladora *PostController* é mais complexo, tendo seis métodos, sendo eles, *show, create, post, edit, update* e *destroy.*

O método *show* é responsável por exibir um *post* específico, não precisa estar autenticado para a exibição do *post*, sendo assim, ele é considerado público.

Figura 38 - Método *PostController::show*

```
public function show($id)
{
    $post = Post::findOrFail($id);
    return view('post.index', compact('post'));
}
```

Fonte: Próprio Autor

O método *create* é uma página responsável para a visualização do formulário para a criação de um novo *post,* ele apenas retorna uma *view*.

Figura 39 - Método *PostController::create*

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

Fonte: Próprio Autor

O método *post* é responsável pela criação da postagem ao Banco de Dados, ou seja, ele valida os campos enviados pelo formulário, adiciona a imagem ao diretório definido e salva o caminho no Banco de Dados caso ela exista, cria a postagem relacionando ao usuário logado e então redireciona para o *dashboard* com mensagem de sucesso.

Figura 40 - Método *PostController::post*

```php
public function post(Request $request)
{
    $request->validate([
        'image' => ['nullable', 'image'],
        'title' => ['required', 'min:10', 'max:50'],
        'description' => ['required', 'min:10', 'max:10000'],
    ]);

    $path = null;
    if ($request->has('image')) {
        $path = $request->file('image')->store('posts', ['disk' => 'public']);
    }

    Post::create([
        'image' => $path,
        'user_id' => Auth::id(),
        'title' => $request->title,
        'description' => $request->description,
    ]);

    return redirect()->route('dashboard')->with('success', true);
}
```

Fonte: Próprio Autor

As regras de validação são que a imagem pode ser *null*, ou seja, não obrigatório e precisa ser do tipo de imagem (formato jpg, jpeg, etc). O título é obrigatório, precisa ter de no mínimo 10 caracteres e no máximo 50.A descrição é obrigatória, deve ter no mínimo 10 caracteres e no máximo 10000. E no final da requisição, o método redireciona para a rota “*dashboard*” com mensagem de sucesso.

O método *edit* busca a postagem com base no *id* passado pela *URI*, válida caso o usuário logado (utilizando a *Facade Auth*) é o responsável por aquele post, caso não seja aborta a requisição com o código 403, caso seja, ele retorna a página de edição do *post*.

Figura 41 - Método *PostController::edit*

```php
public function edit($id)
{
    $post = Post::findOrFail($id);

    if ($post->user_id !== Auth::id()) {
        abort(403);
    }

    return view('post.update', compact('post'));
}
```

Fonte: Próprio Autor

O método *update* é muito semelhante ao método *create*, a sua diferença está em que ao invés de criar, ele válida o *id* passado pela URI e então atualiza aquela postagem específica.

Figura 42 - Método *PostController::update*

```php
public function update(Request $request, $id)
{
    $request->validate([
        'image' => ['nullable', 'image'],
        'title' => ['required', 'min:10', 'max:50'],
        'description' => ['required', 'min:10', 'max:10000'],
    ]);

    $path = null;
    if ($request->has('image')) {
        $path = $request->file('image')->store('posts', ['disk' => 'public']);
    }

    $post = Post::findOrFail($id);

    if ($post->user_id !== Auth::id()) {
        abort(403);
    }

    $post->update([
        'image' => $path,
        'user_id' => Auth::id(),
        'title' => $request->title,
        'description' => $request->description,
    ]);

    return redirect()->route('dashboard')->with('success', true);
}
```

Fonte: Próprio Autor

O último método é o *destroy*, que é responsável por apenas deletar o registro no Banco de Dados, mas para que isso ocorra, é necessário validar se o *post* solicitado a remoção é do usuário logado.

Figura 43 - Método *PostController::destroy*

```php
public function destroy($id)
{
    $post = Post::findOrFail($id);
    $post->delete();

    if ($post->user_id !== Auth::id()) {
        abort(403);
    }

    return redirect()->route('dashboard')->with('success', true);
}
```

Fonte: Próprio Autor

4.12.5 Criação das Views

Alguns dos métodos das controladores retornam *views*, essas visões precisam ser criadas de acordo com o parâmetros passados no helper *view*, exemplo a *view* "*post.update*", é preciso criar o arquivo "*update.blade.php*" dentro do diretório "*resources/views/post*".

A primeira *view* é responsável por exibir todos os *posts* vindo do *HomeController*, sendo assim precisamos no arquivo "*views/home/index.blade.php*" chamar o *foreach* ou o *forelse* caso seja necessário validar se não existe nenhum *post* dentro da variável "*$posts"*.

Figura 44 - *View home.index*

```
@forelse($posts as $post)
    <a href="{{route('posts.show', ['id' => $post->id])}}">
        <div class="bg-white rounded shadow overflow-hidden">
            @if($post->image)
                <img src="{{ url('storage/' . $post->image)}}" alt="{{$post->title}}" class="w-full h-40 object-cover">
            @else
                <div class="w-full h-40 bg-gray-200 flex items-center justify-center">
                    <p class="text-gray-400">Nenhuma imagem cadastrada.</p>
                </div>
            @endif
            <div class="p-4 flex flex-col justify-between h-36">
                <div>
                    <h1 class="font-bold">{{$post->title}}</h1>
                    <p class="mb-4">{{ Str::limit($post->description, 30, ' (...)') }}</p>
                </div>
                <div class="flex justify-between text-xs text-gray-600">
                    <div>
                        <p class="font-semibold">Criado por:</p>
                        <p>{{$post->user->name}}</p>
                    </div>
                    <div>
                        <p class="font-semibold">Última atualização:</p>
                        <p>{{$post->updated_at->format('d/m/Y H:i:s')}}</p>
                    </div>
                </div>
            </div>
        </div>
    </a>
@empty
    <div>Nenhum post cadastrado.</div>
@endforelse
```

Fonte: Próprio Autor

As outras *views* são responsáveis da exibição de criação, atualização ou visualização de um *post* específico.

Figura 45 - *View post.index*

```
<x-guest-layout>
    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8 my-4">

    <h1 class="font-bold text-lg underline">
        {{ $post->title }}
    </h1>

    <div class="bg-white rounded shadow overflow-hidden">
        @if($post->image)
            <img src="{{ url('storage/' . $post->image)}}" alt="{{$post->title}}" class="w-full h-96 object-cover">
        @endif
        <div class="p-4 flex flex-col justify-between h-36">
            <div>
                <p class="mb-4">
                    {{$post->description}}
                </p>
            </div>
            <div class="flex justify-between text-xs text-gray-600">
                <div>
                    <p class="font-semibold">Criado por:</p>
                    <p>{{$post->user->name}}</p>
                </div>
                <div>
                    <p class="font-semibold">Última atualização:</p>
                    <p>{{$post->updated_at->format('d/m/Y H:i:s')}}</p>
                </div>
            </div>
        </div>
    </div>

    <a href="{{route('home')}}" class="text-gray-800 my-4 block">
        Voltar
    </a>
</x-guest-layout>
```

Fonte: Próprio Autor

Figura 46 - *View post.create*

```
<x-app-layout>
    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 leading-tight">
            {{ __('Criar Post') }}
        </h2>
    </x-slot>

    @if ($errors->any())
        <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
            <div class="bg-white text-red-400 rounded p-4 my-4">
                <ul>
                    @foreach ($errors->all() as $error)
                        <li>{{ $error }}</li>
                    @endforeach
                </ul>
            </div>
        </div>
    @endif

    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
        <div class="bg-white p-4 rounded mt-4">
            <form action="{{route('posts.post')}}" method="POST" enctype="multipart/form-data">
                @csrf
                <div class="flex flex-col mb-4">
                    <label for="image">Imagem:</label>
                    <input name="image" type="file" accept="image">
                </div>
                <div class="flex flex-col mb-4">
                    <label for="title">Título</label>
                    <input type="text" name="title" class="border border-gray-400 rounded" value="{{old('title')}}">
                </div>
                <div class="flex flex-col mb-4">
                    <label for="description">Descrição</label>
                    <textarea name="description" class="border border-gray-400 rounded resize-none h-48">{{old('description')}}</textarea>
                </div>
                <div class="flex justify-end">
                    <button type="submit" class="bg-green-500 px-4 py-2 text-sm text-white rounded ">Salvar</button>
                </div>
            </form>
        </div>
    </div>
</x-app-layout>
```

Fonte: Próprio Autor

Figura 47 - *View post.update*

```
<x-app-layout>
    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 leading-tight">
            {{ __('Atualizar Post:') }} {{$post->title}}
        </h2>
    </x-slot>

    @if ($errors->any())
        <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
            <div class="bg-white text-red-400 rounded p-4 my-4">
                <ul>
                    @foreach ($errors->all() as $error)
                        <li>{{ $error }}</li>
                    @endforeach
                </ul>
            </div>
        </div>
    @endif

    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
        <div class="bg-white p-4 rounded mt-4">
            <form action="{{route('posts.update', ['id' => $post->id])}}" method="POST" enctype="multipart/form-data">
                @csrf
                @method('PUT')
                <div class="flex flex-col mb-4">
                    <label for="image">Imagem:</label>
                    <input name="image" type="file" accept="image">
                </div>
                <div class="flex flex-col mb-4">
                    <label for="title">Título</label>
                    <input type="text" name="title" class="border border-gray-400 rounded" value="{{old('title') ?? $post->title}}">
                </div>
                <div class="flex flex-col mb-4">
                    <label for="description">Descrição</label>
                    <textarea name="description" class="border border-gray-400 rounded resize-none h-48">{{old('description') ?? $post->description}}</textarea>
                </div>
                <div class="flex justify-end">
                    <button type="submit" class="bg-green-500 px-4 py-2 text-sm text-white rounded ">Salvar</button>
                </div>
            </form>
        </div>
    </div>
</x-app-layout>
```

Fonte: Próprio Autor

Na *view* de criação e atualização de uma postagem, é notável reparar que há a validação de que se há erros de uma *request* e os *helpers old*, no qual imprime o valor da *request* anterior que falhou e o *helper route*, que busca a rota com base no arquivo de rotas que será definido em seguida.

4.12.6 Rotas

No arquivo *"routes/web.php"* vamos é preciso definir quais são as URI e para qual controladora ela irá apontar, junto com a *middleware* de obrigatoriedade de autenticação e seus nomes.

Figura 48 - Rotas dos *posts*

```
Route::get('/', [HomeController::class, 'index'])->name('home');
Route::get('/posts/create',     [PostController::class, 'create'])->middleware(['auth'])->name('posts.create');
Route::get('/posts/{id}',       [PostController::class, 'show'])->name('posts.show');
Route::post('/posts',           [PostController::class, 'post'])->middleware(['auth'])->name('posts.post');
Route::get('/posts/{id}/edit',  [PostController::class, 'edit'])->middleware(['auth'])->name('posts.edit');
Route::put('/posts/{id}',       [PostController::class, 'update'])->middleware(['auth'])->name('posts.update');
Route::delete('/posts/{id}',    [PostController::class, 'destroy'])->middleware(['auth'])->name('posts.destroy');
```

Fonte: Próprio Autor

As rotas com a *middleware "auth"* faz com que seja obrigatório que o usuário esteja logado, e cada rota tem um nome para que quando chamado o *helper route*, se passe o nome da rota ao invés do caminho.

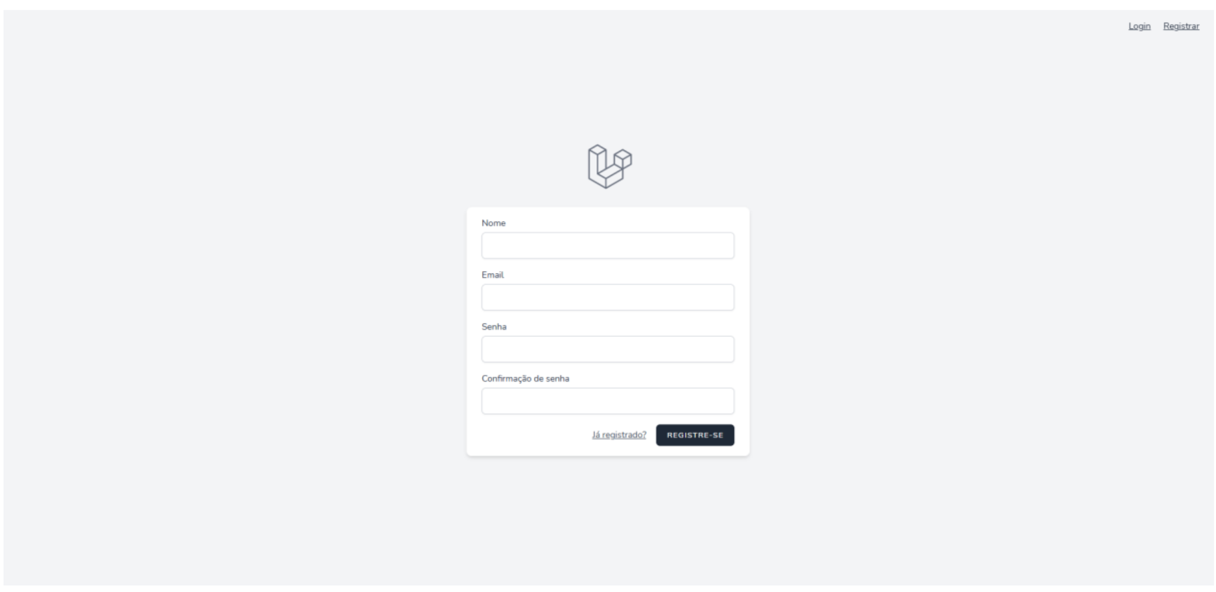
4.12.7 Projeto Final e Telas

Como resultado final, podemos ter a página de entrar e de cadastro, junto com a página que exibe todos os *posts* e todo o CRUD (*Create*, *Read*, Update e *Delete*) da postagem para o usuário logado.

Figura 49 - Página sem nenhuma postagem

Fonte: Próprio Autor

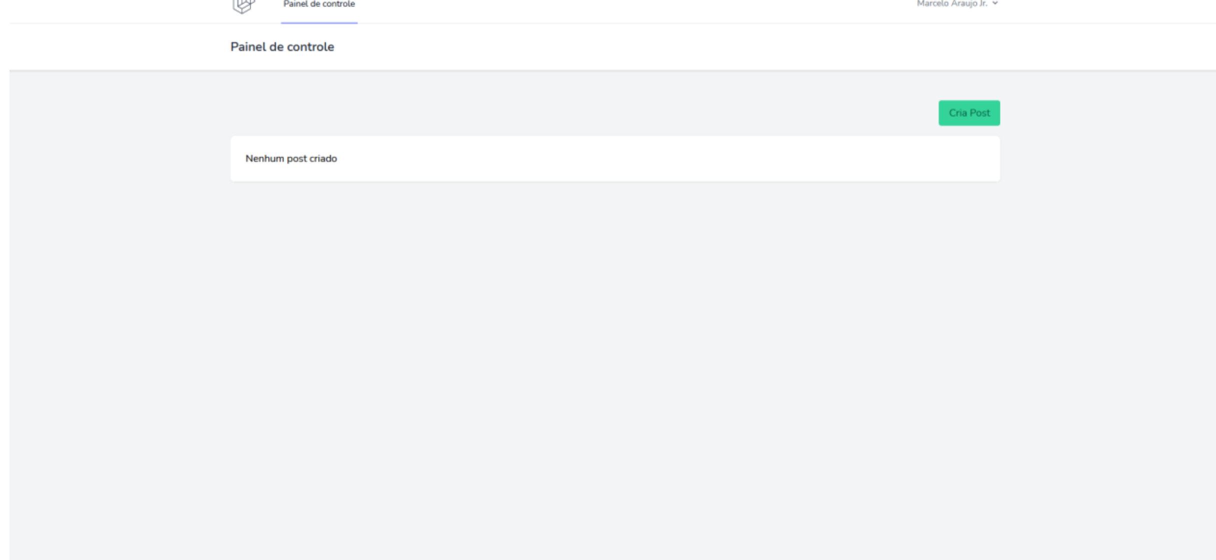
Figura 50 - Página de Registro



Fonte: Próprio Autor

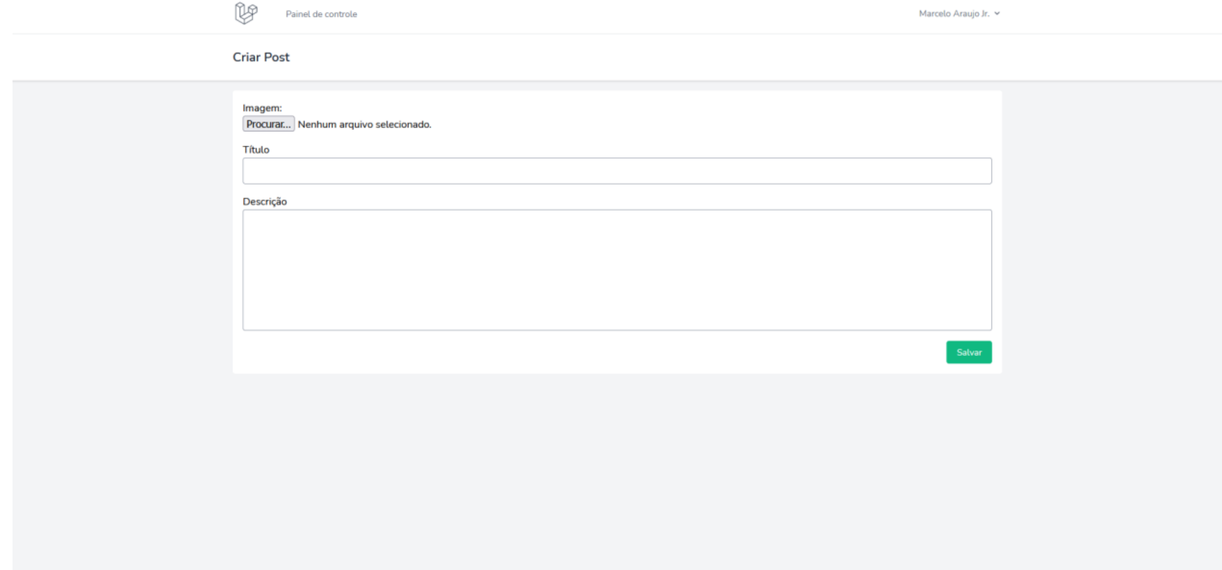
Após realizar o registro e realizar o acesso ao sistema, será apresentado para a página de todos os *posts* do usuário logado.

Figura 51 - Página de todos os *posts* do usuário logado



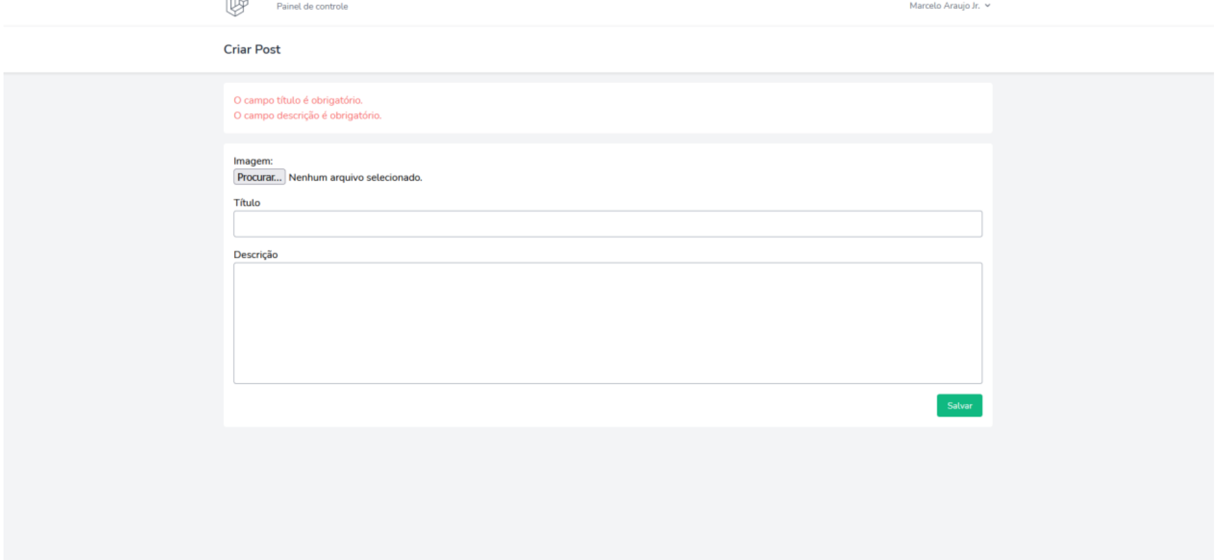
Fonte: Próprio Autor

Figura 52 - Página de cadastro de postagem



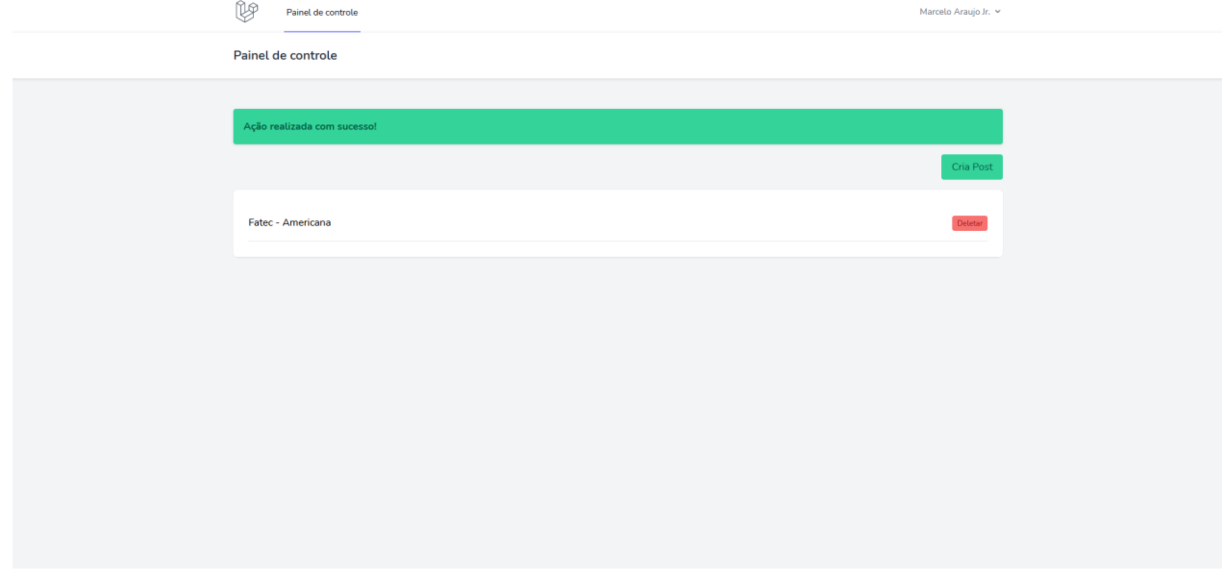
Fonte: Próprio Autor

Figura 53 - Página de cadastro de postagem com erro



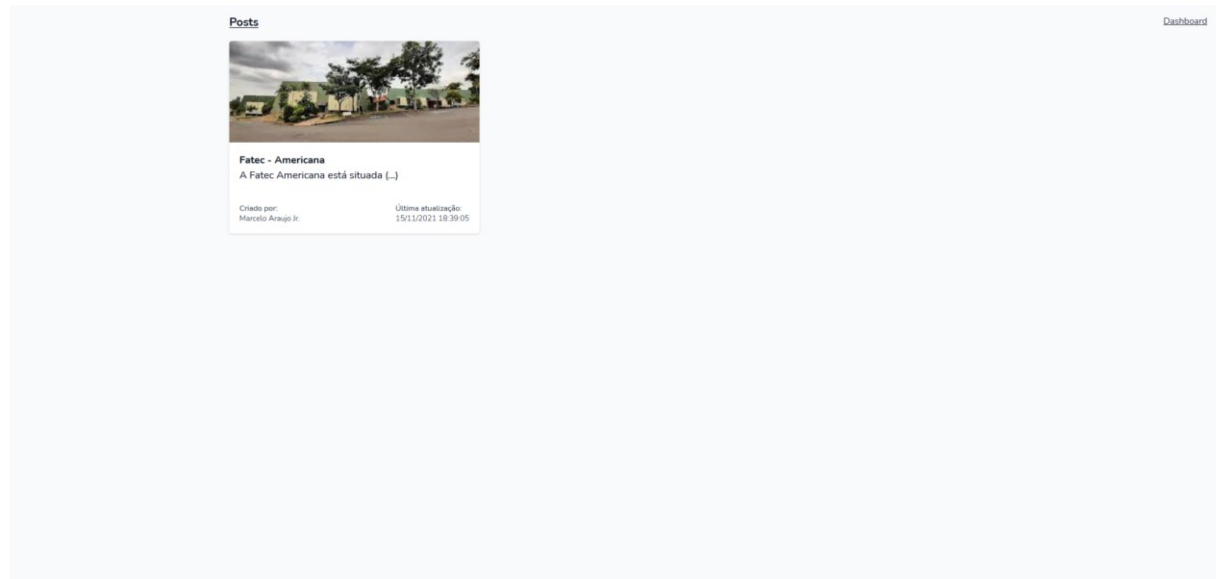
Fonte: Próprio Autor

Figura 54 - Criação de uma postagem com sucesso



Fonte: Próprio Autor

Figura 55 - *Home* com postagens



Fonte: Próprio Autor

Figura 56 - Imagem interna da postagem

Fonte: Próprio Autor

Todo o Projeto pode ser encontrado no endereço github.com/mlajx/laravel-demo.

5 CONCLUSÃO

A procura por ferramentas que agilizem o desenvolvimento de aplicações e que mantenham o código padronizado cresce cada vez mais e o Laravel possui uma função muito importante, trazendo diversas ferramentas que contribuem para a otimização de Projetos simples ou complexos.

O Laravel é um excelente e completo *Framework* que permite a criação de vários sistemas, tendo suporte para a criação de diversos Projetos.

Apesar de sua estrutura ser MVC, ele pode se estender a muito mais, permitindo que cada Projeto tenha sua própria estrutura de código, como por exemplo a criação de uma camada de repositório, que é responsável pela chamada das *Models* para a comunicação com o Banco de Dados, e os Controladores responsáveis apenas pelas chamadas dos Repositórios.

Mas para que possamos estabelecer um Projeto, é necessário desbravar a documentação e outros Projetos de exemplos, de forma a viabilizar o aprendizado para a criação de novos Projetos robustos.

Utilizado por grandes empresas como 9GAG, Pfizer e BBC, o Laravel é um *Framework* que a cada vez é mais popular no mundo do PHP. É estimado que o Laravel continue sendo utilizado por diversas empresas e consequentemente, seus códigos permanecerão em constantes atualizações.

REFERÊNCIAS BIBLIOGRÁFICAS

BEAN, M. **Laravel 5 Essentials**. [s.l.] Packt Publishing Ltd, 2015.

Blade Templates - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/blade#blade-directives>. Acesso em: 11 nov. 2021.

Como criar um pacote para o Laravel. Disponível em: <https://imasters.com.br/php/como-criar-um-pacote-para-o-laravel>. Acesso em: 31 out. 2021.

Conheça 9 Frameworks para fazer site em PHP. **Programadores Brasil**, 9 jun. 2020. Disponível em: <https://programadoresbrasil.com.br/2020/06/fazer-site-em-php-dicas-de-frameworks/>. Acesso em: 22 out. 2021

Controllers - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/controllers#introduction>. Acesso em: 11 nov. 2021.

Database: Getting Started - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/database>. Acesso em: 7 nov. 2021.

Database: Migrations - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/migrations#introduction>. Acesso em: 7 nov. 2021.

Database: Query Builder - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/queries>. Acesso em: 10 nov. 2021.

Eloquent: Getting Started - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/eloquent>. Acesso em: 10 nov. 2021.

Eloquent: Relationships - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/eloquent-relationships>. Acesso em: 15 nov. 2021a.

Eloquent: Relationships - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/eloquent-relationships#eager-loading>. Acesso em: 15 nov. 2021b.

Facade. Disponível em: <https://refactoring.guru/pt-br/design-patterns/facade>. Acesso em: 31 out. 2021.

Facades - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/facades#facade-class-reference>. Acesso em: 7 nov. 2021.

GABARDO, A. C. **Laravel para ninjas**. [s.l.] Novatec Editora, 2017.

HTTP Requests - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/requests#introduction>. Acesso em: 11 nov. 2021.

Installation - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/installation#why-laravel>. Acesso em: 22 out. 2021a.

Installation - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/installation#getting-started-on-linux>. Acesso em: 31 out. 2021b.

Installation - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x#choosing-your-sail-services>. Acesso em: 22 out. 2021c.

Introduction - Composer. Disponível em: <https://getcomposer.org/doc/00-intro.md>. Acesso em: 24 out. 2021.

Laravel: Framework PHP • Astronauts Developers. **Astronauts Developers**, 22 set. 2020. Disponível em: <https://astronautsdevelopers.com/laravel/>. Acesso em: 22 out. 2021

Laravel Sail - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/sail#introduction>. Acesso em: 24 out. 2021.

LERDORF, R. et al. **Programming PHP**. [s.l.] O'Reilly Media, Inc., 2002.

MEHTA, C. et al. **MySQL 8 Administrator's Guide: Effective guide to administering high-performance MySQL 8 solutions**. [s.l.] Packt Publishing Ltd, 2018.

Middleware - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/middleware#introduction>. Acesso em: 11 nov. 2021.

Middleware no Laravel (Filtros). **Blog EspecializaTi**, 3 out. 2017. Disponível em: <https://blog.especializati.com.br/middleware-no-laravel-filtros/>. Acesso em: 11 nov. 2021

MILANI, A. **MySQL - Guia do Programador**. [s.l.] Novatec Editora, 2007.

O que é Docker. Disponível em: <https://www.redhat.com/pt-br/topics/containers/what-is-docker>. Acesso em: 24 out. 2021.

O que é Docker · Docker para Desenvolvedores. Disponível em: <https://stack.desenvolvedor.expert/appendix/docker/oquee.html>. Acesso em: 24 out. 2021.

O que é padrão MVC, Entenda arquitetura de softwares! | Le Wagon. Disponível em: <https://www.lewagon.com/pt-BR/blog/o-que-e-padrao-mvc>. Acesso em: 10 nov. 2021.

PHP: Introdução - Manual. Disponível em: <https://www.php.net/manual/pt_BR/language.types.intro.php>. Acesso em: 21 out. 2021.

PHP: O que é o PHP - Manual. Disponível em: <https://www.php.net/manual/pt_BR/intro-whatis.php>. Acesso em: 21 out. 2021.

PHP: O que o PHP pode fazer - Manual. Disponível em: <https://www.php.net/manual/pt_BR/intro-whatcando.php>. Acesso em: 21 out. 2021.

Request Lifecycle - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/lifecycle#http-console-kernels>. Acesso em: 31 out. 2021a.

Request Lifecycle - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/lifecycle#finishing-up>. Acesso em: 31 out. 2021b.

Routing - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/routing>. Acesso em: 10 nov. 2021.

SEGERS, J. **Laravel MongoDB**. [s.l: s.n.].

Service Container - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/container#introduction>. Acesso em: 31 out. 2021.

Service Providers - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/providers>. Acesso em: 31 out. 2021a.

Service Providers - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/providers#writing-service-providers>. Acesso em: 31 out. 2021b.

Starter Kits - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/starter-kits#laravel-breeze>. Acesso em: 15 nov. 2021.

Validation - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/validation#introduction>. Acesso em: 11 nov. 2021.

Views - Laravel - The PHP Framework For Web Artisans. Disponível em: <https://laravel.com/docs/8.x/views>. Acesso em: 11 nov. 2021.

What is a Container | App Containerization | Docker. Disponível em: <https://www.docker.com/resources/what-container>. Acesso em: 24 out. 2021.