

O guia abaixo tem por objetivo ensinar o desenvolvimento de uma aplicação no back-end em [Laravel 8.54](#) com o banco de dados MySQL. O principal objetivo é criar uma API que realiza CRUD (Create, Read, Update, Delete) de País e Universidades. Este guia é para o ambiente Ubuntu 20.04.

Ambiente Linux Ubuntu 20.04

Instalando o Composer/PHP

Uma aplicação Laravel necessita de várias bibliotecas auxiliares para funcionar. Um sistema, para rodar uma aplicação laravel, necessita das bibliotecas listadas abaixo:

- PHP >= 7.3.0
- XML PHP Extension
- PDO PHP Extension
- JSON PHP Extension
- CType PHP Extension
- BCMath PHP Extension
- OpenSSL PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension

Considerando que o servidor foi instalado com as bibliotecas necessárias, também vamos precisar instalar o composer. Ele é utilizado para a criação do projeto em Laravel. Os passos para a instalação do composer são apresentados abaixo.

Instalando o composer

Caso você tenha tido sucesso na instalação do Composer na Semana 2, você pode pular esta etapa. Caso contrário, prossiga com a execução dos comandos.

Deslocar para a home do sistema operacional:

```
cd ~
```

A seguir, executar o comando abaixo para instalar o composer utilizando o curl:

```
curl -sS https://getcomposer.org/installer -o composer-setup.php
```

Na sequência, execute o seguinte comando e instale o composer globalmente na sua máquina/host:

```
sudo php composer-setup.php --install-dir=/usr/local/bin --filename=composer
```

Desta forma, o composer está instalado na sua máquina. Ao terminar basta você executar o comando abaixo:

```
composer -V
```

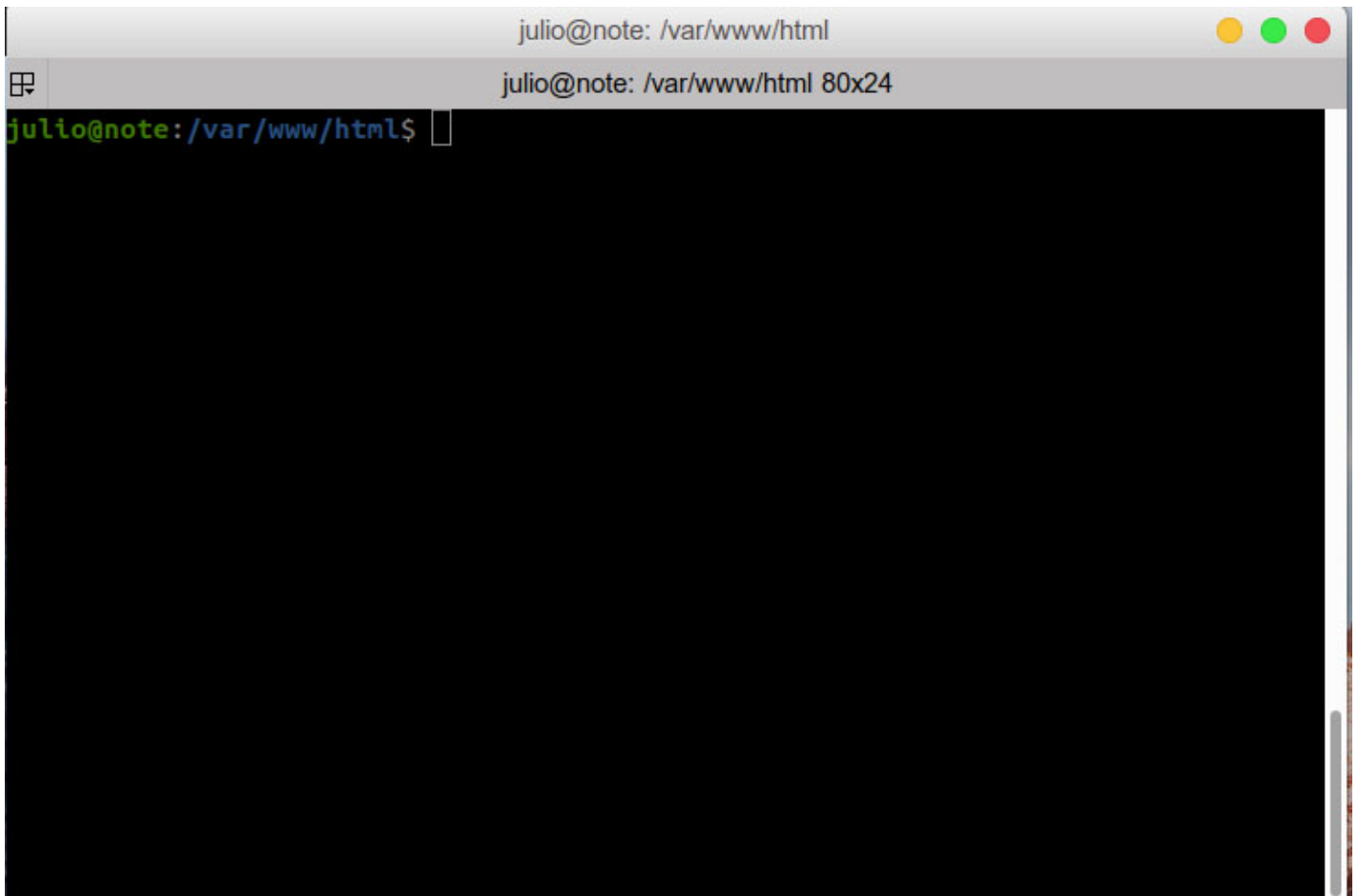
e visualizar a versão do composer instalada.

Criando projeto Laravel

A nossa aplicação deverá ficar armazenada em **/var/www/html/**. Ainda no terminal, acesse a pasta em que o projeto deve ser criado e execute o comando:

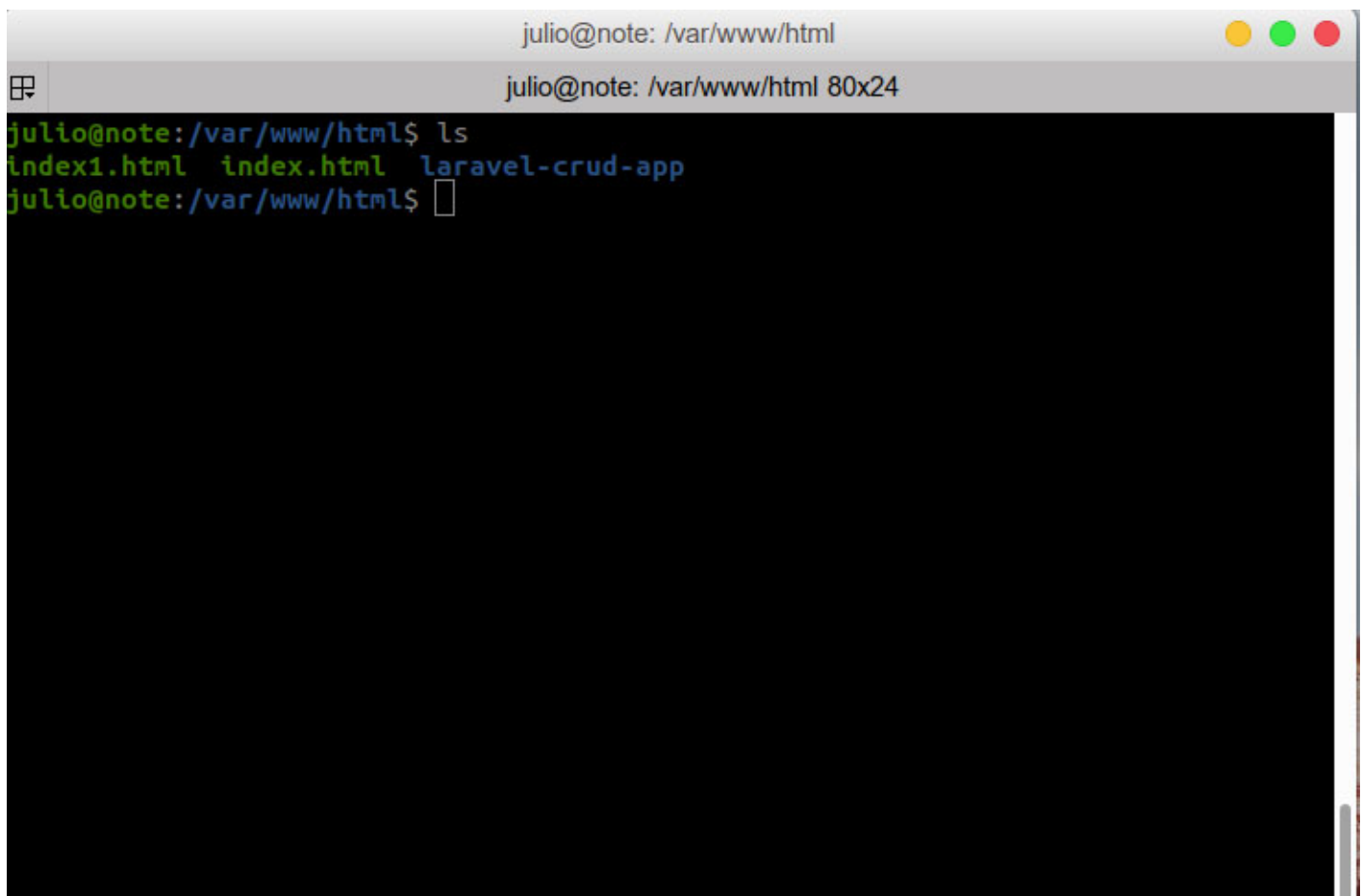
```
composer create-project laravel/laravel --prefer-dist laravel-crud-app
```

E teremos uma saída como a mostrada a seguir:

A screenshot of a terminal window. The title bar at the top reads 'julio@note: /var/www/html' and has three window control buttons (yellow, green, red) on the right. Below the title bar, the terminal shows the prompt 'julio@note: /var/www/html 80x24' followed by a new line and the prompt 'julio@note:/var/www/html\$' with a cursor. The rest of the terminal area is black and empty.

```
julio@note: /var/www/html
julio@note: /var/www/html 80x24
julio@note:/var/www/html$
```

Para criar um projeto laravel, atente-se para o **laravel-crud-app**, uma vez que esse é o nome do projeto que deve ser substituído antes de executar o comando.



```
julio@note: /var/www/html
julio@note: /var/www/html 80x24
julio@note:/var/www/html$ ls
index1.html  index.html  laravel-crud-app
julio@note:/var/www/html$
```

Após executar esse comando, basta acessar a pasta do projeto:

```
cd laravel-crud-app
```

E verificar a versão do laravel instalada:

```
php artisan -V
```

Configurando o MySQL

Nesse passo, será apresentado como se faz a conexão entre o banco de dados MySQL na aplicação do aplicativo do Laravel. Lembre-se que as variáveis de nome do banco de dados, usuário e senha do banco devem ser ajustadas de acordo com as configurações presentes na máquina e o nome do banco de dados que você criou. Assumo que vocês já possuem o banco de dados instalado, com usuário e senhas definidas. Neste material, meu usuário é o root com a senha que defini na instalação do MySQL.

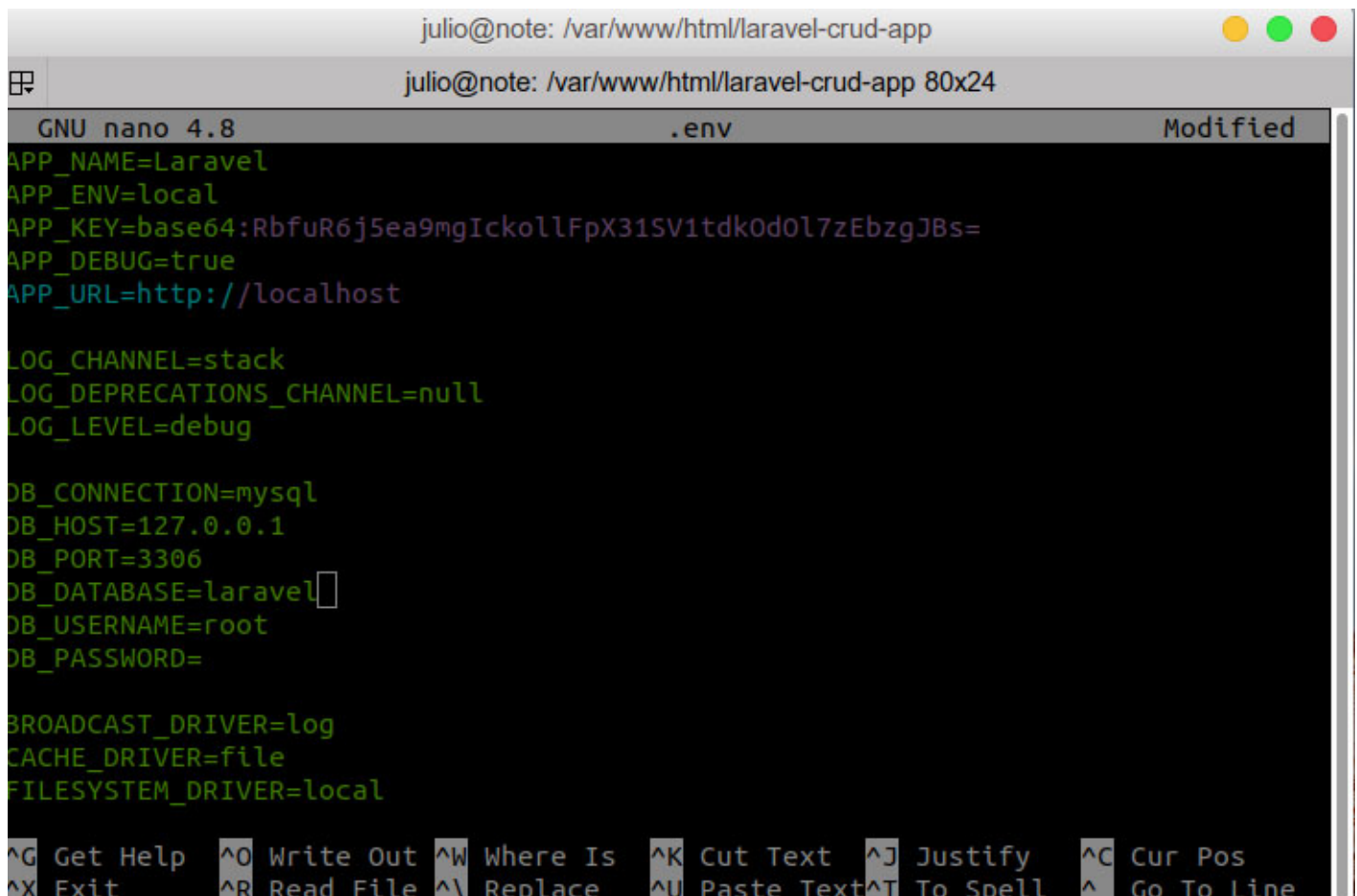
A configuração e a criação do banco de dados devem ser realizadas antes de executar as migrações do Laravel. Você deve criar um banco e dar um nome que reflete o que o sistema fará. Para este guia, consideramos a instalação e configuração do Laravel (Semana 2) e também um reforço deste assunto na Semana 4, em que demos o nome do banco de `laravel_db`. Por isso seguimos com este banco já criado anteriormente.

Para configurar o banco de dados no laravel, o arquivo **.env** deve ser editado. Atente-se aqui ao fato de que o arquivo `.env` não sofre **commit** por que as configurações dos serviços são individuais de cada máquina, e normalmente essas variáveis em uma aplicação final são atribuídas diretamente por variáveis de ambiente do SO (Sistema Operacional). Se o arquivo `.env` não existir no projeto baixado de um repositório do git, basta copiar o `.env.example` e atribuir as respectivas variáveis necessárias.

Considerando que você esteja no diretório corrente do projeto, abra o arquivo `.env` com o seguinte comando:

```
pico .env
```

O resultado é mostrado na tela abaixo:

A screenshot of a terminal window with a dark background. The window title is 'julio@note: /var/www/html/laravel-crud-app'. The terminal shows the nano 4.8 editor editing the .env file. The file content includes configuration for APP_NAME, APP_ENV, APP_KEY, APP_DEBUG, APP_URL, LOG_CHANNEL, LOG_DEPRECATIONS_CHANNEL, LOG_LEVEL, DB_CONNECTION, DB_HOST, DB_PORT, DB_DATABASE, DB_USERNAME, DB_PASSWORD, BROADCAST_DRIVER, CACHE_DRIVER, and FILESYSTEM_DRIVER. The bottom of the screen shows nano editor shortcuts like '^G Get Help', '^O Write Out', etc.

```
julio@note: /var/www/html/laravel-crud-app
julio@note: /var/www/html/laravel-crud-app 80x24
GNU nano 4.8 .env Modified
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:RbfuR6j5ea9mgIckollFpX31SV1tdkOd0l7zEbzgJBs=
APP_DEBUG=true
APP_URL=http://localhost

LOG_CHANNEL=stack
LOG_DEPRECATIONS_CHANNEL=null
LOG_LEVEL=debug

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=

BROADCAST_DRIVER=log
CACHE_DRIVER=file
FILESYSTEM_DRIVER=local

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Paste Text ^T To Spell ^_ Go To Line
```

Após realizar as mudanças no .env o código deve ficar como abaixo:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel_db
DB_USERNAME=root
DB_PASSWORD=COLOQUE_SUA_SENHA_AQUI
```

com o resultado também mostrado na imagem a seguir:

```
julio@note: /var/www/html/laravel-crud-app
julio@note: /var/www/html/laravel-crud-app 80x24
GNU nano 4.8 .env Modified
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:RbfuR6j5ea9mgIckollFpX31SV1tdk0d0l7zEbzgJBs=
APP_DEBUG=true
APP_URL=http://localhost

LOG_CHANNEL=stack
LOG_DEPRECATIONS_CHANNEL=null
LOG_LEVEL=debug

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel_db
DB_USERNAME=root
DB_PASSWORD=

BROADCAST_DRIVER=log
CACHE_DRIVER=file
FILESYSTEM_DRIVER=local

[ Read 52 lines ]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Paste Text ^T To Spell ^_ Go To Line
```

Salve as modificações com o comando: **CTRL+X** e depois digite **w**. Pronto, o arquivo foi salvo!!!

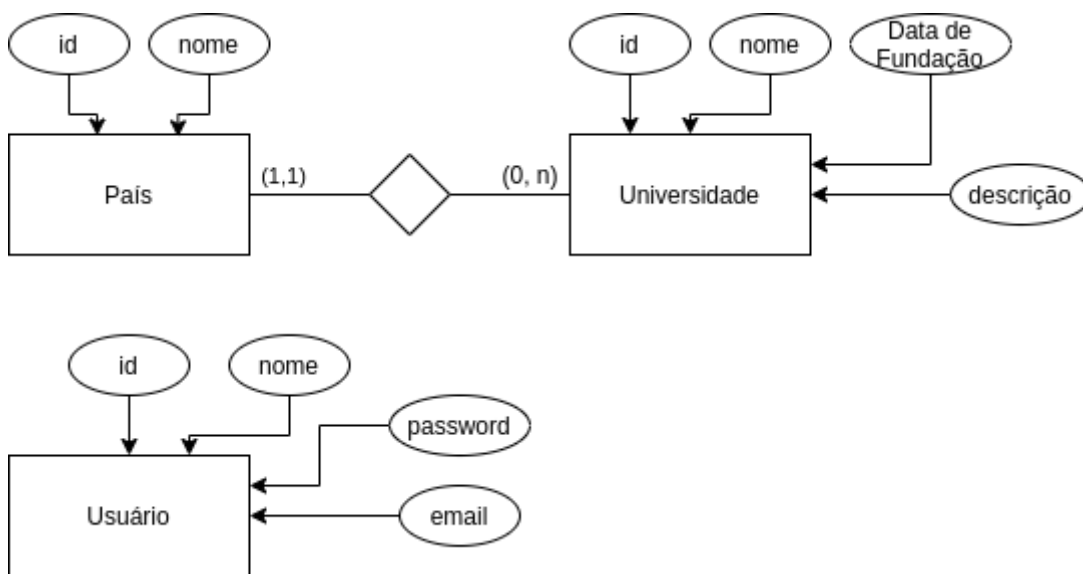
Além disso é interessante executar o seguinte comando para limpar o cache que pode existir na aplicação:

```
php artisan config:clear
```

Atente-se para uma configuração adicional que deve ser feita caso o sistema macOS seja utilizado para o desenvolvimento.

```
DB_SOCKET=/Applications/MAMP/tmp/mysql/mysql.sock
```

Como já discutido na semana anterior, e apenas para relembrar, vamos utilizar o modelo de entidade e relacionamento conforme a figura abaixo:



Este modelo contém três tabelas: `Usuário`, `País` e `Universidade`. Além disso, cada `Universidade` está vinculada a um país e dessa forma existe uma chave estrangeira de país em `Universidade`. Como base neste exemplo do modelo, podemos exibir como são mapeadas as entidades relacionadas em Laravel.

Criando uma Migração e um Modelo

Até este ponto, criamos o banco de dados e configuramos o banco de dados adicionando as credenciais no arquivo `env`. Em seguida, aprenderemos como definir a migração adicionando as propriedades de dados na tabela MySQL. Uma migração Laravel é a definição de uma ação que altera o banco de dados como criação de tabela, alteração de coluna, exclusão de base de dados, etc. Veja que não faremos a criação do banco de dados de forma tradicional (entrando no `mysql`, ou usando outra ferramenta como o `phpMyAdmin`), pois o framework Laravel está preparado para que, por meio de funções, os comandos de criação sejam feitos rapidamente para auxiliar o desenvolvedor. A vantagem é que o processo de desenvolvimento se torna mais ágil.

Precisamos criar um arquivo de modelo e migração para criar as migrações. Para esta etapa, execute no terminal os comandos a seguir, na pasta **laravel-crud-app**.

```
php artisan make:model País -m
php artisan make:model Universidade -m
```

Dentro do projeto Laravel foram criados arquivos no diretório: `database/migrations`. Por padrão o Laravel criou arquivos com o seguinte padrão:

- `timestamp_create_pais_table.php`
- `timestamp_create_universidades_table.php`

No arquivo de migração de país nós temos o seguinte código:

```
<?php

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

class CreatePaisTable extends Migration
{
    /**
```

```

    * Run the migrations.
    *
    * @return void
    */
public function up()
{
    Schema::create('pais', function (Blueprint $table) {
        $table->bigIncrements('id');
        $table->timestamps();
    });
}

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

```

De acordo com o que foi definido no modelo de dados de País, devemos adicionar o campo `nome`. Digite no terminal: `pico timestamp_create_universidades_table.php` (verifique o número criado para o timestamp e substitua-o antes de chamar o pico). Para realizar essa adição basta colocar o seguinte código na função de `up()`:

```

$table->string('nome');

```

O resultado final deve ficar como na tela a seguir:

```
julio@note: /var/www/html/laravel-crud-app/database/migrations
julio@note: /var/www/html/laravel-crud-app/database/migrations 80x24
GNU nano 4.8      2021_10_20_155735_create_pais_table.php
<?php

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

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

[ Read 32 lines ]
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Paste Text ^T To Spell  ^_ Go To Line
```

Já o mapeamento da migração para Universidade necessita da adição de outros campos no método `up()`. Digite no terminal: `pico timestamp_create_universidades_table.php` (verifique o número criado para o timestamp e substitua-o antes de chamar o pico). Esses campos são apresentados abaixo:

```
$table->string('nome');
$table->string('descricao', 600);
$table->date('dt_fundacao');
$table->foreignIdFor(Pais::class); // pais_id
```

O resultado final deve ficar como na tela a seguir:


```
julio@note: /var/www/html/laravel-crud-app/database/migrations
julio@note: /var/www/html/laravel-crud-app/database/migrations 80x24
GNU nano 4.8 2021 10 20 155757 create universidades table.php
class CreateUniversidadesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('universidades', function (Blueprint $table) {
            $table->id();
            $table->string('nome');
            $table->string('descricao', 600);
            $table->date('dt_fundacao');
            $table->foreignIdFor(Pais::class); // pais_id
            $table->timestamps();
        });
    }
}
```

Aqui definimos um campo do tipo **data** para a data da fundação da universidade, e definimos um tamanho de 600 para o **varchar** do campo no MySQL. E, por fim, uma **chave estrangeira** para país é adicionada à universidade. Você deve também adicionar a linha no arquivo da classe Universidade:

```
use App\Models\Pais;
```

logo após o `<?php`, indicando a importação da classe pais no migrate de universidade.

A classe de universidade ao final dessa mudança fica da seguinte forma:

```
<?php

use App\Models\Pais;
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateUniversidadesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('universidades', function (Blueprint $table) {
```

```

        $table->id();
        $table->string('nome');
        $table->string('descricao', 600);
        $table->date('dt_fundacao');
        $table->foreignIdFor(Pais::class); // pais_id
        $table->timestamps();
    });
}

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

```

E a classe de Pais ficará da seguinte forma:

```

<?php

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

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

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {

```

```
        Schema::dropIfExists('pais');
    }
}
```

Dentro das migrações, há duas funções que são herdadas de `Illuminate\Database\Migrations\Migration`; . Essas funções são `up()` e `down()` . A função `up()` é executada quando a migração for realizada com o comando `php artisan migrate` . No caso do código acima, a migração cria uma tabela chamada `país` , que tem atributos `id` , `nome` e os timestamps padrões para data/hora da criação e da última edição. O método `down()` existe para reverter as ações realizadas pelo método `up()` , ou seja, aqui estão os comandos para dropar tabela, colunas e outros dados.

Ao terminar de configurar as migrações, basta executar no terminal, dentro da pasta **laravel-crud-app**, o comando `php artisan migrate` , e as tabelas serão criadas.

A seguir será preciso entrar em `app/Models` , considerando que você esteja no diretório raiz da aplicação: **laravel-crud-app**. Execute o comando

```
cd app/Models
```

Agora vamos editar os modelos. Primeiro vamos fazer este procedimento no arquivo `Pais.php`. Abra o arquivo com o comando:

```
pico Pais.php
```

O código inicial do modelo será igual a esse:

```
namespace App\Models;

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

class Pais extends Model
{
    use HasFactory;
}
```

Deve ser adicionada a variável protegida `$fillable` . O laravel usa o modelo `mass-assignable` que permite mapear quais os valores serão mapeados para o cadastro diretamente no banco de dados. Isso significa que, se passar um array com o valor determinado no `$fillable` , ele será mapeado para uma coluna na tabela mapeada para o modelo. Por exemplo, adicionando o `$fillable` no modelo `Pais`, o conteúdo final do arquivo ficará como a seguir:

```
<?php
namespace App\Models;

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

class Pais extends Model
{
    use HasFactory;
```

```
protected $fillable = ['nome'];
}
```

Salve o arquivo com o comando: **CTRL+X** e seguida digite **w** quando perguntado se deseja salvar. Em seguida tecle **ENTER**. Pronto!!!

Para o código de Universidade use o comando:

```
pico Universidade.php
```

e adicione o seguinte conteúdo abaixo de use HasFactory :

```
protected $fillable = [
    'nome',
    'descricao',
    'dt_fundacao',
    'pais_id'
];

protected $hidden = ['pais_id'];
```

O arquivo final deve ficar como segue:

```
<?php

namespace App\Models;

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

class Universidade extends Model
{
    use HasFactory;

    protected $fillable = [
        'nome',
        'descricao',
        'dt_fundacao',
        'pais_id'
    ];

    protected $hidden = ['pais_id'];
}
```

Salve o arquivo com o comando **CRTL+X** e depois digite **w** para confirmar. A seguir, tecle **ENTER**. Pronto!!!

Veja que é possível ver as colunas mapeadas em `$fillable` . Há também uma nova variável chamada de `$hidden`. Esta variável `$hidden` é responsável por ocultar o campo quando existir uma consulta que recupera os dados dessa tabela. Essa variável é importante para quando se quer ocultar informações do serviço.

Agora, vamos apresentar o mapeamento da migração da entidade Universidade. Conforme o modelo do banco de dados definido, precisamos dos campos: `id`, `nome`, `descricao`, `dt_fundacao`, `pais_id` e os campos de data de criação e edição. O campo `pais_id` é uma chave-estrangeira de país. Vá para o diretório `database/migrations` e abra o arquivo `_create_universidades_table.php`.

```
cd /database/migrations

pico _create_universidades_table.php
```

Veja como ele ficou:

```
<?php

use App\Models\Pais;
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateUniversidadesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('universidades', function (Blueprint $table) {
            $table->id();
            $table->string('nome');
            $table->string('descricao', 600);
            $table->date('dt_fundacao');
            $table->foreignIdFor(Pais::class); // pais_id
            $table->timestamps();
        });
    }

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

Uma chave-estrangeira é mapeada normalmente como `nome_da_table_id`. Esse é o padrão adotado no Laravel.

Controllers

Nos controladores estão os códigos lógicos da aplicação e as ações para cada modelo. Um novo controller pode ser criado executando o código:

```
php artisan make:controller PaisController --resource
php artisan make:controller UniversidadeController --resource
```

Os comandos anteriores criam respectivamente os controllers: `app/Http/Controllers/PaisController.php` e `app/Http/Controllers/UniversidadeController.php`. O comando `--resource` cria por padrão sete métodos definidos para cada um dos controllers que criamos:

- `index()`: lista todos os dados do controller.
- `create()`: retorna a página de formulário para criação do controller. (não utilizado em API's)
- `store()`: usado para criar um novo registro no banco de dados.
- `show()`: usado para exibir um registro específico no banco de dados.
- `edit()`: retorna a página de formulário para alterar do controller. (não utilizado em API's)
- `update()`: usado para atualizar um registro específico no banco de dados.
- `destroy()`: usado para deletar um registro no banco de dados.

Mais especificações sobre [Controllers no Laravel](#) podem ser visualizados no link da documentação oficial.

Métodos do Controller Pais

Vá para o diretório `app/Http/Controllers/` e em seguida procure pelo arquivo **PaisController.php**:

```
cd app/Http/Controllers/
```

Abra um dos arquivos, por exemplo, o `PaisController.php`, e em seguida o `UniversidadeController.php`, e verá que os conteúdos dos métodos estão vazios. O próximo passo é editar esses métodos em ambos os arquivos para lidar com o escopo da nossa aplicação.

Vamos tratar de alguns métodos no `PaisController.php`. O método de `index()` por exemplo é bem simples, pois a ideia é que ele retorne tudo do controller específico. Edite o método como segue:

```
public function index()
{
    return response()->json(
        Pais::all()
    );
}
```

Na sequência, vamos editar o método `store()`. Este, é utilizado para criar um novo registro. Nele, devemos validar os dados que foram enviados na requisição POST, realizar o cadastro e por fim retornar o dado cadastrado e o status 201. O código do store deve ficar como mostrado abaixo:

```
public function store(Request $request)
{
    $storeData = Validator::make($request->all(), [
        'nome' => 'required|max:255'
    ]);

    if ($storeData->fails()) {
```

```

        return response()->json($storeData, 422);
    }

    return response()->json(
        Pais::create($request->all()),
        201
    );
}

```

Vamos prosseguir, com o método `show()`. Ele é utilizado para recuperar um registro específico. O código deste método para recuperar um único registro deve ficar como mostrado abaixo:

```

public function show($id)
{
    return response()->json(
        Pais::findOrFail($id)
    );
}

```

Vamos então lidar com o método `update()`, o qual é utilizado para atualizar um registro. Nele são recebidos os dados e o 'id' do dado que deve ser atualizado. Atualize o código como mostrado abaixo:

```

public function update(Request $request, $id)
{
    $updateData = Validator::make($request->all(), [
        'name' => 'required|max:255',
        'descricao' => 'required|max:255',
        'dt_fundacao' => 'required',
    ]);

    if ($updateData->fails()) {
        return response()->json($updateData, 422);
    }

    $universidade = Universidade::whereId($id)->update($request->all());
    return response()->json($universidade);
}

```

Por fim, chegamos ao método `destroy()`, cujo objetivo é remover um registro específico de acordo com o id informado. Atualize este método de forma que ele fique como mostrado abaixo:

```

public function destroy($id)
{
    $universidade = Universidade::findOrFail($id);
    $universidade->delete();
    return response()->json(null, 204);
}

```

Métodos do Controller Universidade

Vá para o diretório `app/Http/Controllers/` e em seguida procure pelo arquivo **UniversidadeController.php**:

```
cd app/Http/Controllers/
```

Vamos tratar de alguns métodos no `UniversidadeController.php`. O método de `index()` por exemplo é bem simples, pois a ideia é que ele retorne tudo do controller específico. Edite o método `index()`, que deve ficar como segue:

```
public function index()
{
    return response()->json(
        Universidade::all()
    );
}
```

Na sequência, vamos editar o método `store()`. Este, é utilizado para criar um novo registro. Nele devemos validar os dados que foram enviados na requisição POST, realizar o cadastro e por fim retornar o dado cadastrado e o status 201. O código do store deve ficar como mostrado abaixo:

```
public function store(Request $request)
{
    $storeData = Validator::make($request->all(), [
        'nome' => 'required|max:255',
        'descricao' => 'required|max:600',
        'dt_fundacao' => 'required',
        'pais_id' => 'required'
    ]);

    if ($storeData->fails()) {
        return response()->json($storeData, 422);
    }

    return response()->json(
        Universidade::create($request->all()),
        201
    );
}
```

Vamos prosseguir, com o método `show()`. Ele é utilizado para recuperar um registro específico. O código deste método, para recuperar um único registro deve ficar como mostrado abaixo:

```
public function show($id)
{
    return response()->json(
        Universidade::findOrFail($id)
    );
}
```

Vamos então lidar com o método `update()`, o qual é utilizado para atualizar um registro. Nele são recebidos os dados e o 'id' do dado que deve ser atualizado. Atualize o código como mostrado abaixo:


```

public function update(Request $request, $id)
{
    $updateData = Validator::make($request->all(), [
        'nome' => 'required|max:255',
        'descricao' => 'required|max:600',
        'dt_fundacao' => 'required',
        'pais_id' => 'required'
    ]);

    if ($updateData->fails()) {
        return response()->json($updateData, 422);
    }

    $universidade = Universidade::whereId($id)->update($request->all());
    return response()->json($universidade);
}

```

Por fim, chegamos ao método `destroy()`, cujo objetivo é remover um registro específico de acordo com o id informado. Atualize este método de forma que ele fique como mostrado abaixo:

```

public function destroy($id)
{
    $universidade = Universidade::findOrFail($id);
    $universidade->delete();
    return response()->json(null, 204);
}

```

Routes

As rotas aqui são mapeadas nas funções dos controllers da maneira 1 para 1. Assim, uma rota `/países` invoca uma função de `PaisController`, por exemplo. Como estamos trabalhando como um serviço e não diretamente com aplicações web integradas, vamos criar as rotas no diretório: `routes/api.php`. Como vamos criar um serviço devemos usar os métodos do HTTP: `GET`, `POST`, `PUT` e `DELETE` para interagir com a base de dados, obedecendo desta forma as definições de RESTful.

Primeiro, acesse o diretório `routes`:

```
cd routes
```

Em seguida, abra o arquivo `api.php`:

```
pico api.php
```

O conteúdo apresentado é o mostrado a seguir:

```

?php

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;

```

```

/*
|-----|
| API Routes
|-----|
|
| Here is where you can register API routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| is assigned the "api" middleware group. Enjoy building your API!
|
*/

Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});

```

Vamos primeramente importar as duas classes: PaisController e UniversidadeController, e em seguida acrescentar as rotas que são primordiais para a nossa aplicação. O código ficará como a seguir:

```

<?php

use App\Http\Controllers\PaisController;
use App\Http\Controllers\UniversidadeController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;

/*
|-----|
| API Routes
|-----|
|
| Here is where you can register API routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| is assigned the "api" middleware group. Enjoy building your API!
|
*/

Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});

Route::get('/paises', [PaisController::class, 'index']);
Route::get('/paises/{id}', [PaisController::class, 'show']);
Route::post('/paises', [PaisController::class, 'store']);
Route::put('/paises', [PaisController::class, 'update']);
Route::delete('/paises', [PaisController::class, 'destroy']);

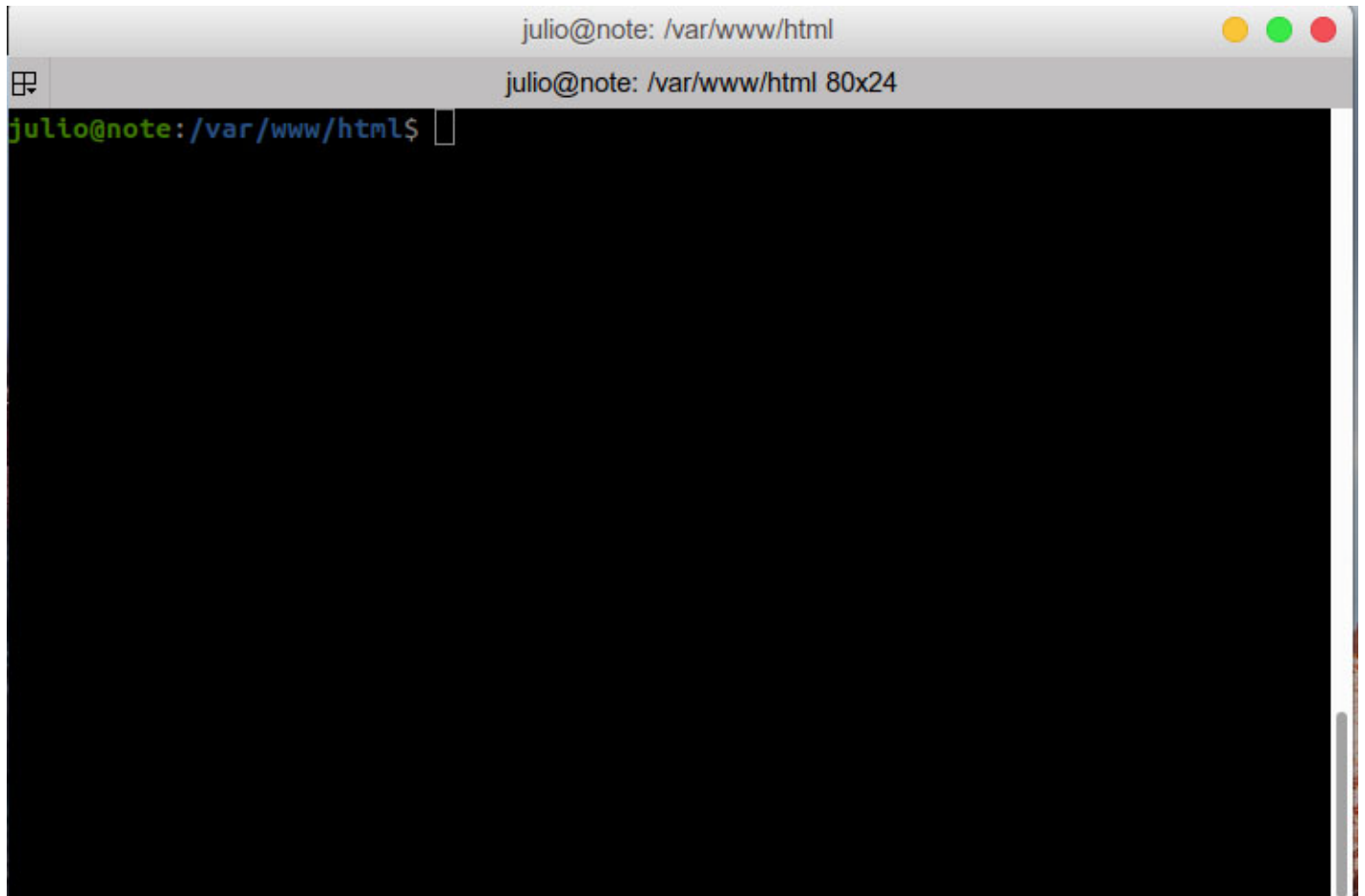
Route::get('/universidades', [UniversidadeController::class, 'index']);
Route::get('/universidades/{id}', [UniversidadeController::class, 'show']);
Route::post('/universidades', [UniversidadeController::class, 'store']);
Route::put('/universidades', [UniversidadeController::class, 'update']);
Route::delete('/universidades', [UniversidadeController::class, 'destroy']);

```

É possível observar o mapeamento de uma classe e uma função no seguinte formato: [PaisController::class, 'index'] . Isso quer dizer que, quando invocamos por exemplo a rota /países , chamamos a função index do controller PaisController. As outras rotas seguem a mesma lógica.

Ajustes do Virtual Host do Apache

Precisamos configurar um host virtual no Apache para podermos acessar nossa aplicação de forma independente dentro do Apache. Veja que desenvolvemos toda a aplicação na pasta /var/www. O nome da nossa aplicação é laravel-crud-app e fica localizada dentro da pasta WWW, que é a raiz do servidor Web Apache. Para confirmar, veja a tela a seguir:



Na sequência, temos que procurar pela pasta sites-available, que contém arquivos de configurações de hosts virtuais. Estando no terminal do Linux, execute o comando a seguir:

```
cd /etc/apache2/sites-available/
```

A seguir, crie um arquivo chamado: meus-projetos.conf dentro da pasta sites-available:

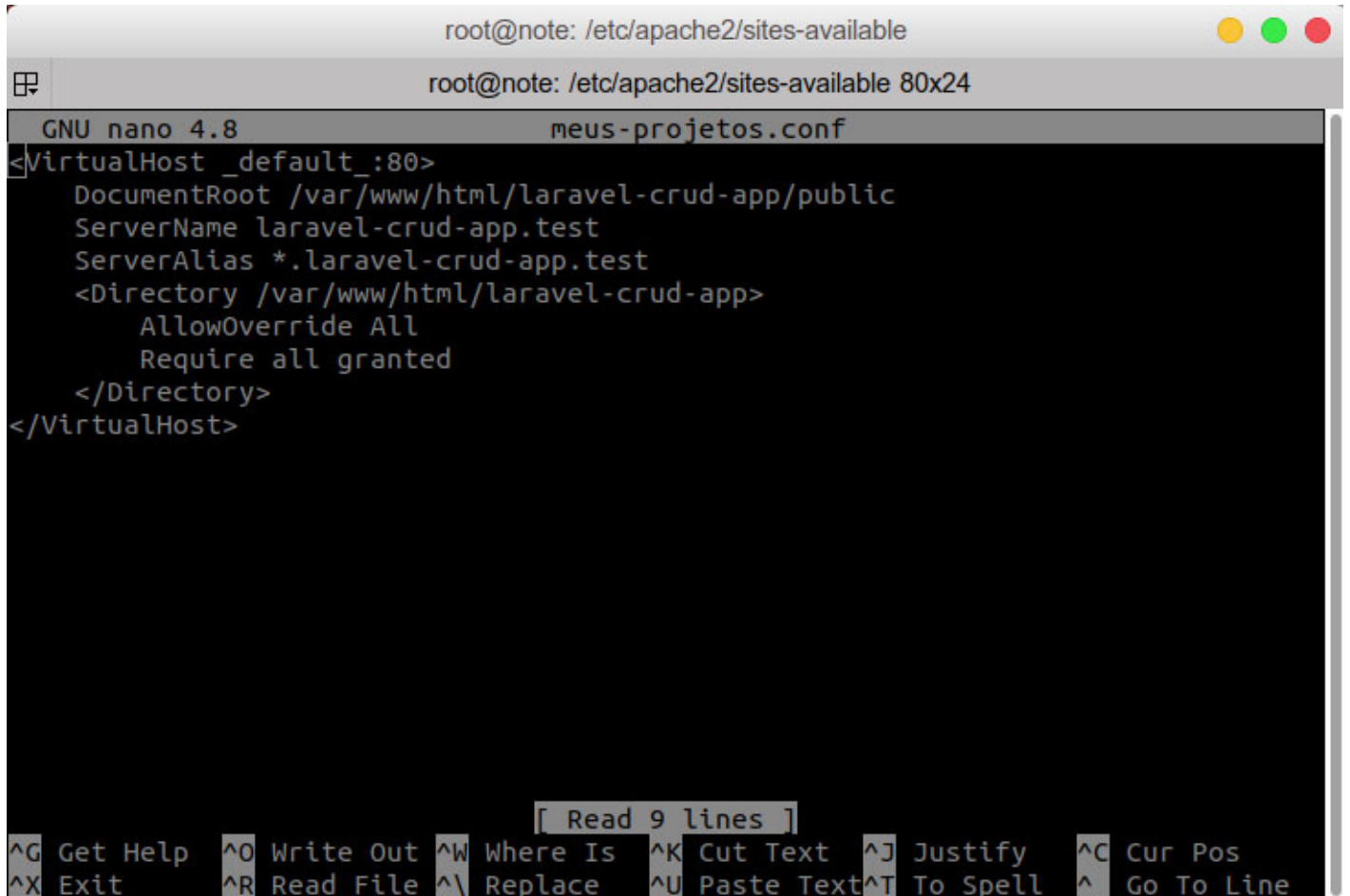
```
pico meus-projetos.conf
```

Dentro deste arquivo, inclua o conteúdo abaixo:

```
<VirtualHost *:80>
ServerName localhost
ServerAdmin admin@example.com
DocumentRoot /var/www/html/laravel-crud-app/public
<Directory /var/www/html/laravel-crud-app>
```

```
AllowOverride All
</Directory>
ErrorLog ${APACHE_LOG_DIR}/error.log
CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

O resultado deve se parecer com a tela abaixo:



```
root@note: /etc/apache2/sites-available
root@note: /etc/apache2/sites-available 80x24
GNU nano 4.8 meus-projetos.conf
<VirtualHost _default_:80>
  DocumentRoot /var/www/html/laravel-crud-app/public
  ServerName laravel-crud-app.test
  ServerAlias *.laravel-crud-app.test
  <Directory /var/www/html/laravel-crud-app>
    AllowOverride All
    Require all granted
  </Directory>
</VirtualHost>
[ Read 9 lines ]
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Paste Text ^T To Spell  ^_ Go To Line
```

Salve o arquivo com o comando **CRTL+X**, digite **Y** na sequência e tecle **ENTER**. Pronto!!!

O próximo passo é executar na pasta `/etc/apache2/sites-available` o comando a seguir:

```
a2ensite meus-projetos.conf
```

Na sequência, reinicie o Apache com o comando:

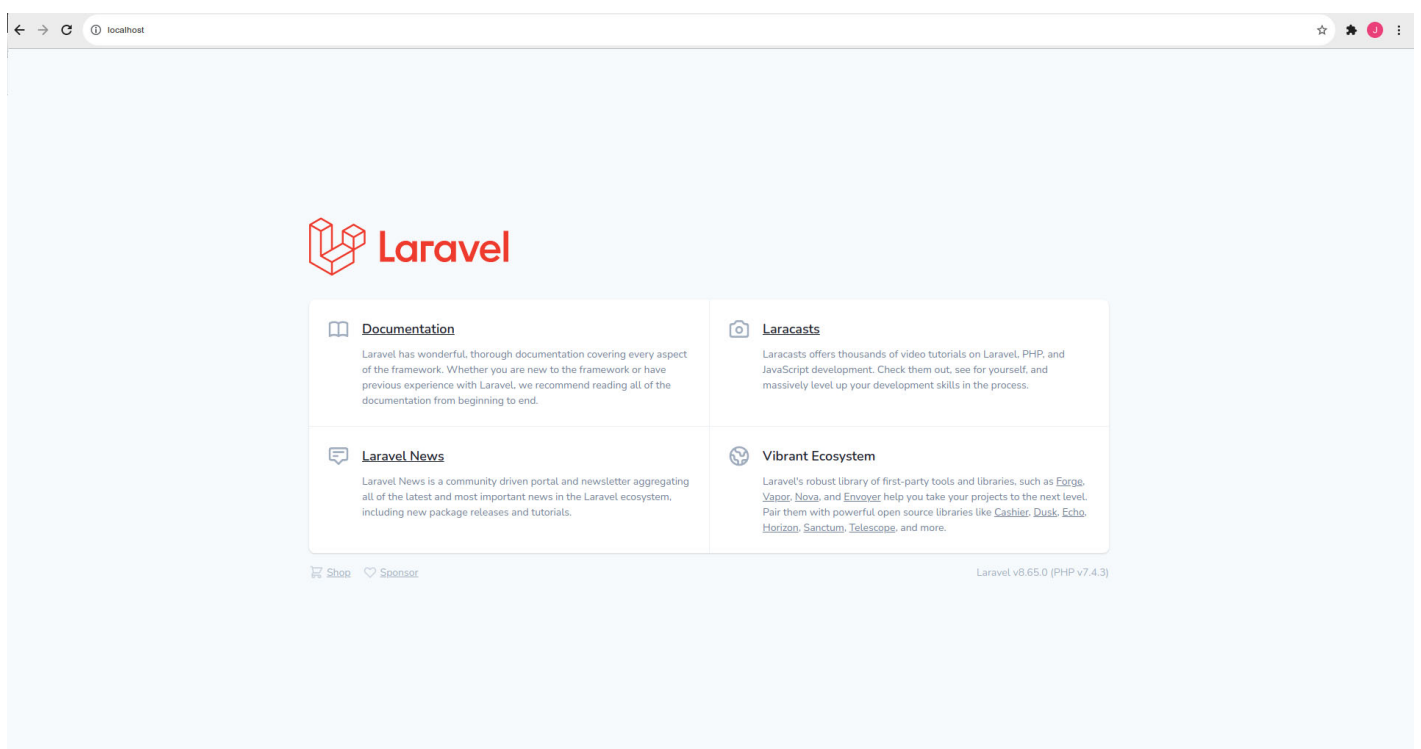
```
systemctl restart apache2
```

```
root@note: /etc/apache2/sites-available
root@note: /etc/apache2/sites-available 80x24
root@note:/etc/apache2/sites-available# systemctl restart apache2
```

Na sequência, abra o Chrome, Firefox, ou navegador de sua preferência e digite na barra de endereços:

`http://localhost`

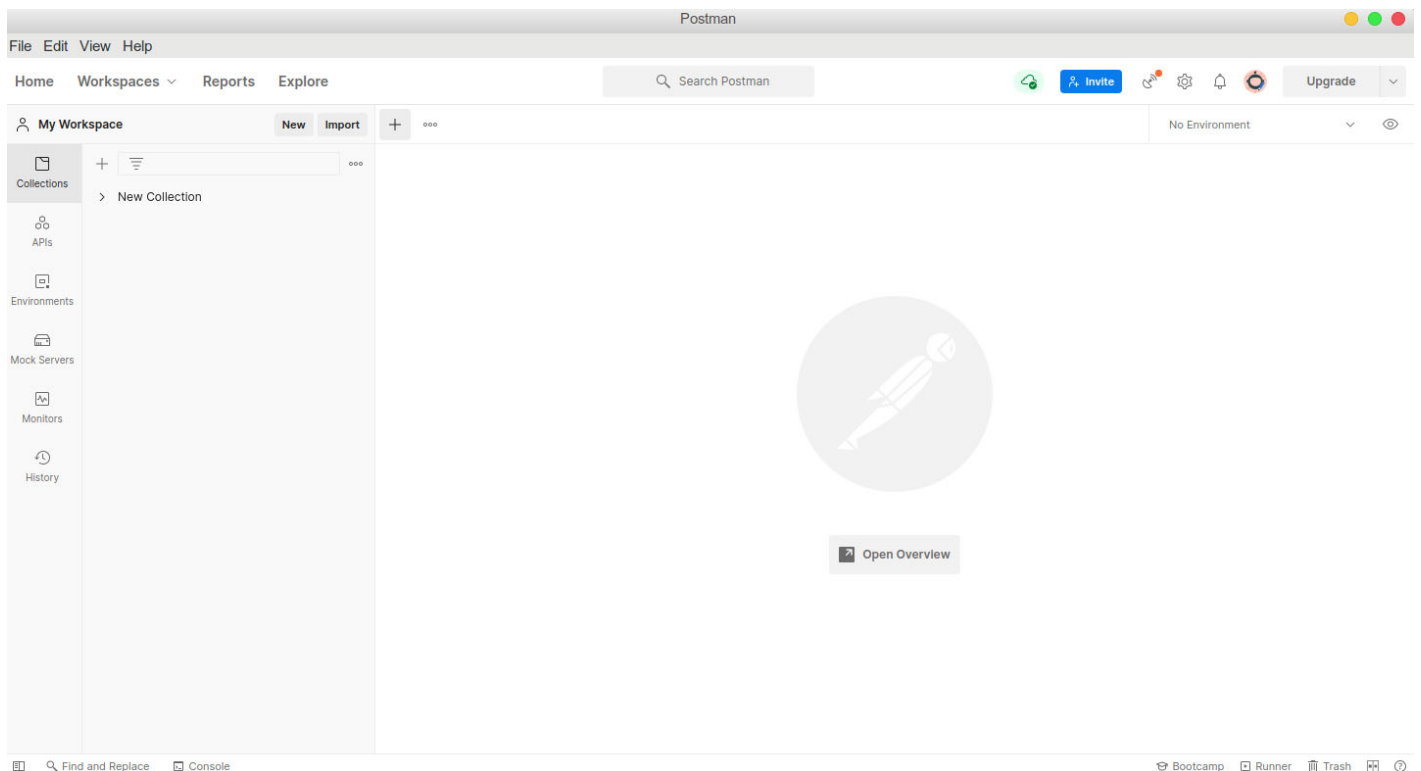
E teremos o nosso virtual host funcionando da forma que foi definida, como destaca a tela abaixo.



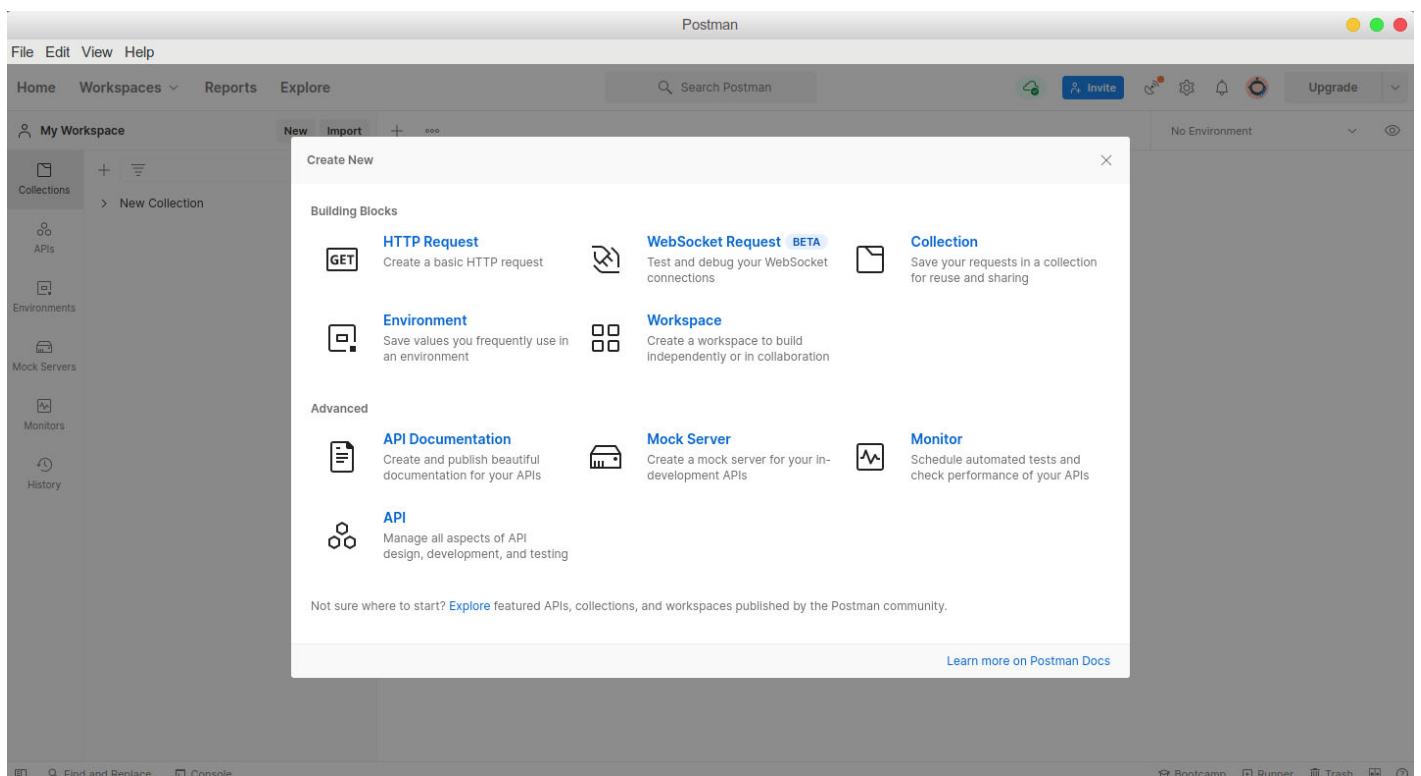
Teste da Aplicação Web

Para testarmos a aplicação Web, vamos utilizar como cliente o software [Postman](#). Escolha a versão para Linux, baixe-o e instale em seu computador. Depois de instalado, execute o postman.

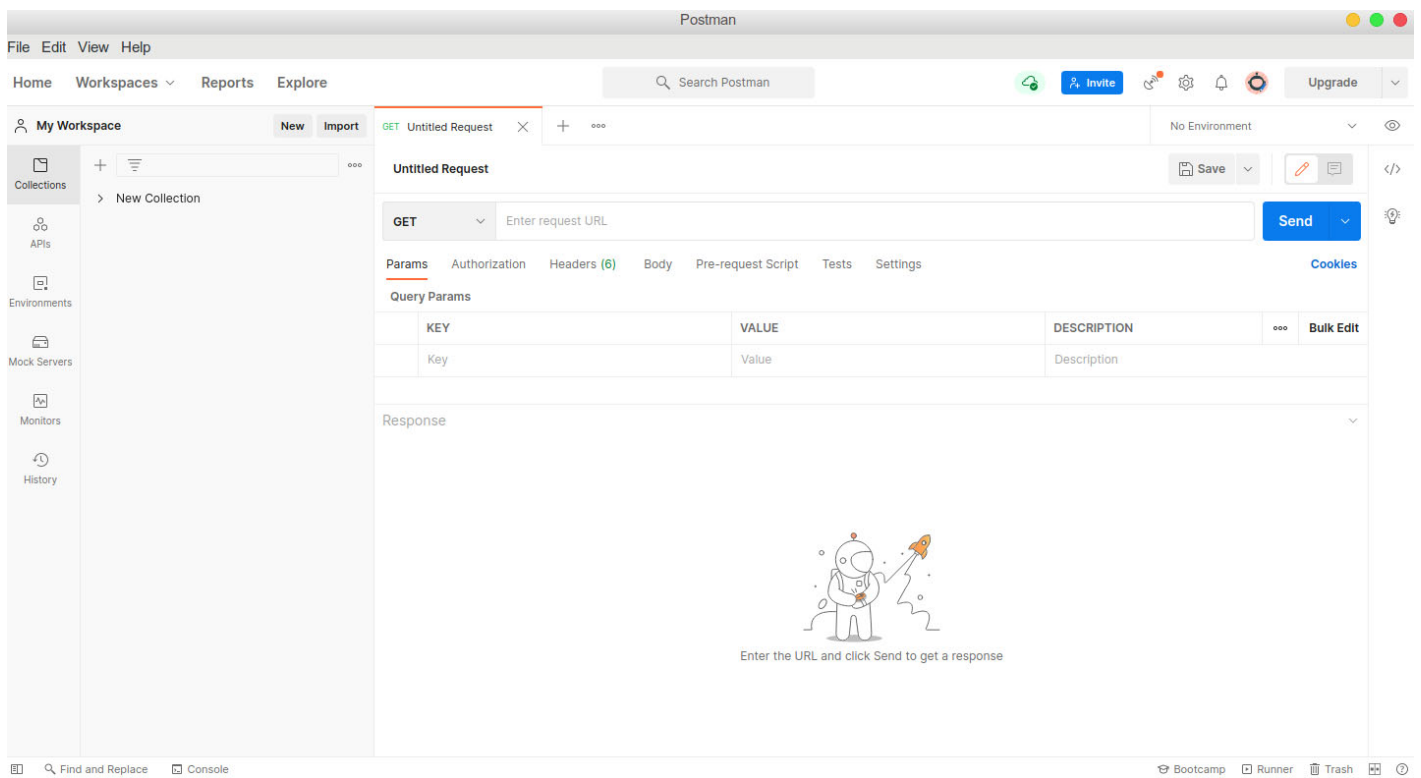
Ao iniciar o postman, voce terá uma tela inicial parecida como a figura abaixo:



Para o primeiro teste, considerando a tela anterior, clique no botão **New**, bem no canto superior esquerdo do painel do postman, e terá como saída a tela a seguir:



Na sequência, clique em **HTTP Request** e teremos acesso à tela para enviarmos comandos HTTP para popular ou recuperar dados da aplicação no banco de dados do MySQL. O resultado é mostrado abaixo:



O próximo passo é ajustar alguns comandos no postman.

Escolha a opção POST

Digite na barra de endereço ao lado do POST, o seguinte: `http://localhost/api/paises`

Setar a opção body

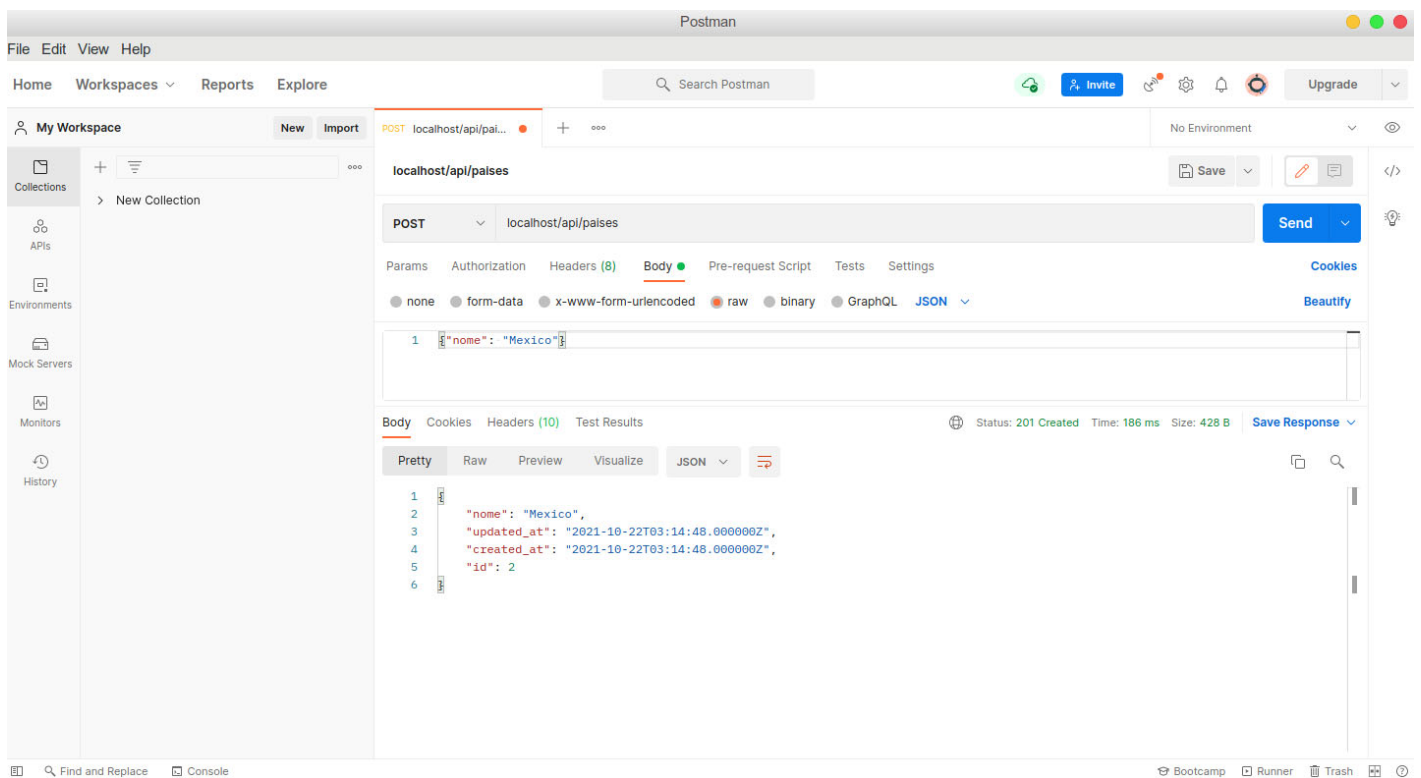
Em seguida escolher raw

Depois mude a opção Text para json

Na sequência, incluir o conteúdo no Body:

```
{"nome": "México" }
```

E clique no botão SEND. A saída final é dada pela tela abaixo:



Observe que já havia um registro inserido na base de dados, e por isso o id do país México é 2. Vamos verificar como acessar a informação cadastrada por meio do GET, mas no navegador. Digite no navegador o comando:

http://localhost/api/paises

E verá que a saída final é a recuperação da base de dados de 2 países cadastrados.



Para Praticar

Considere as outras rotas que codificamos anteriormente, como: recuperar um país pelo id, inserir uma universidade, recuperar uma universidade pelo id, apagar universidades, atualizar universidades pelo id, bem como remover os países e também atualizar um país pelo id. Utilize o postman para incluir, apagar e atualizar os dados na aplicação. Fiquem atentos ao conteúdo que deve ser passado no corpo da requisição HTTP. Por exemplo, no caso das universidades, ao criar uma requisição HTTP passado um conteúdo no formato json, você não pode se esquecer que todos os parâmetros precisam ser preenchidos, pois definimos isso como obrigatórios no esquema da tabela Universidades lá no banco de dados.

É isso aí pessoal. Espero que aproveitem este material e que ele sirva de apoio para você se aprofundar no desenvolvimento de soluções que envolvem o back-end de uma aplicação Web. Até a próxima!!!