2024

Starter kit de Laravel

GUIA MEGA BÁSICA DE LARAVEL 11

MTRO MIGUEL CASTRO

Contenido

1 TEORIA

1.1 Objetivo

El objetivo de esta guía es proporcionar a los desarrolladores un recurso comprensivo y práctico para la creación de aplicaciones web robustas y escalables utilizando el framework Laravel. A través de esta guía, los desarrolladores aprenderán los fundamentos de Laravel, así como técnicas avanzadas y mejores prácticas para maximizar la eficiencia, seguridad y rendimiento de sus proyectos. Esta guía abarca desde la configuración inicial del entorno de desarrollo, hasta la implementación de características avanzadas, con ejemplos claros y detallados, así como consejos para resolver problemas comunes y optimizar el flujo de trabajo

1.2 ¿Qué es Laravel?

Laravel es un framework de desarrollo web de código abierto basado en PHP, diseñado para facilitar y agilizar el proceso de creación de aplicaciones web robustas y mantenibles. Fue creado por Taylor Otwell y lanzado por primera vez en junio de 2011. Laravel sigue el patrón de arquitectura MVC (Model-View-Controller), que separa la lógica de la aplicación en tres componentes interconectados para una mayor organización y escalabilidad.

1.2.1 Características Principales de Laravel

1. **Eloquent ORM**: Un sistema ORM (Object-Relational Mapping) elegante y sencillo para interactuar con bases de datos. Permite a los desarrolladores trabajar con bases de datos mediante modelos de Eloquent en lugar de consultas SQL directas.
2. **Motor de Plantillas Blade**: Un motor de plantillas potente y ligero que proporciona características como herencia de plantillas y secciones, haciendo que la creación de vistas sea más sencilla y organizada.
3. **Rutas (Routing)**: Un sistema de rutas simple y flexible que facilita la definición de rutas para tu aplicación. Laravel ofrece rutas web, API, y rutas de consola.
4. **Middleware**: Un mecanismo para filtrar solicitudes HTTP, útil para implementar autenticación, autorización y otras funcionalidades antes de que la solicitud llegue a los controladores.
5. **Migrations**: Un sistema para versionar y gestionar la base de datos, permitiendo a los desarrolladores crear y modificar tablas y otros componentes de la base de datos de manera controlada y reproducible.
6. **Sistema de Autenticación**: Laravel proporciona un sistema de autenticación y autorización listo para usar, con opciones personalizables para gestionar usuarios y roles.

7. **Comandos Artisan**: Una interfaz de línea de comandos (CLI) llamada Artisan, que incluye una variedad de comandos útiles para tareas comunes como la creación de controladores, modelos, migraciones y más.
8. **Colas de Trabajos (Queues)**:Un sistema para gestionar trabajos en segundo plano, como el envío de correos electrónicos o el procesamiento de tareas largas, mejorando la eficiencia y la experiencia del usuario.
9. **Broadcasting**: Facilita la implementación de características en tiempo real, como notificaciones o actualizaciones de contenido en aplicaciones web mediante WebSockets.
10. **Modularidad**: Laravel es altamente modular, permitiendo la integración de paquetes adicionales a través de Composer, el gestor de dependencias de PHP.

1.2.2 Ventajas vs Desventajas

1.2.2.1 Ventajas

- **Ecosistema Rico y Completo**: Laravel ofrece una amplia gama de herramientas integradas, como Eloquent ORM, el motor de plantillas Blade, y el sistema de migraciones, que facilitan el desarrollo y mantenimiento de aplicaciones.
- **Documentación Extensa**: La documentación de Laravel es detallada y fácil de seguir, lo que facilita el aprendizaje y la resolución de problemas.
- **Comunidad Activa**: Una gran comunidad de desarrolladores que contribuyen con paquetes, tutoriales, y soporte a través de foros y redes sociales.
- **Seguridad Integrada**: Mecanismos de seguridad incorporados para proteger contra ataques comunes, como inyección SQL, XSS y CSRF.
- **Artisan CLI**: La interfaz de línea de comandos Artisan ofrece una serie de comandos útiles que automatizan tareas comunes y mejoran la productividad.
- **Eloquent ORM**: Facilita la interacción con bases de datos mediante un sistema ORM intuitivo, permitiendo trabajar con modelos en lugar de consultas SQL.
- **Modularidad y Reusabilidad**: Soporta la creación de paquetes y la integración de bibliotecas externas a través de Composer, lo que promueve la reusabilidad y la modularidad del código.
- **Configuración Fácil**: Laravel proporciona una configuración sencilla y una estructura de directorios clara, lo que facilita el inicio de nuevos proyectos.

- **Flexibilidad y Escalabilidad**: Es adecuado tanto para proyectos pequeños como grandes, y puede escalar con las necesidades de la aplicación.
- **Testing Integrado**: Laravel incluye soporte para pruebas unitarias y funcionales, lo que ayuda a garantizar la calidad del código.

1.2.2.2 Desventajas

- **Curva de Aprendizaje Inicial**: Aunque Laravel es conocido por su facilidad de uso, los principiantes pueden encontrar la curva de aprendizaje inicial algo empinada debido a la cantidad de conceptos y herramientas que ofrece.
- **Rendimiento**: Laravel puede ser más lento en comparación con frameworks más ligeros debido a su naturaleza rica en características y capas de abstracción. Esto puede requerir optimizaciones adicionales para aplicaciones de alto rendimiento.
- **Consumo de Recursos**: Las aplicaciones Laravel pueden consumir más recursos del servidor, lo que podría ser un problema en entornos con recursos limitados.
- **Actualizaciones Frecuentes**: Laravel tiene un ciclo de lanzamiento frecuente, lo que puede llevar a problemas de compatibilidad y la necesidad de actualizar el código regularmente para mantenerse al día con las nuevas versiones.
- **Dependencia de Terceros**: Al usar numerosos paquetes y bibliotecas de terceros, puede haber riesgos de compatibilidad y seguridad si alguno de estos paquetes no se mantiene adecuadamente.
- **Complejidad para Pequeños Proyectos**: Para proyectos muy pequeños o simples, Laravel puede ser excesivo en términos de configuración y características, donde un framework más ligero o una solución sin framework podría ser más adecuada.
- **Desarrollo Backend Enfocado**: Laravel está principalmente orientado al backend, por lo que se necesita integrar otras tecnologías o frameworks para el desarrollo frontend moderno (aunque esto también puede ser una ventaja, dependiendo del contexto del proyecto).

1.3 Requisitos de sistema

- **PHP**: Laravel requiere PHP 8.0 o superior. Asegúrate de tener instalada una versión compatible.
- **Composer**: Es el gestor de dependencias para PHP que Laravel utiliza. Necesitas instalar Composer en tu sistema.

- **Servidor Web**: Aunque Laravel tiene su propio servidor de desarrollo, para producción necesitarás Apache, Nginx u otro servidor web compatible.
- **Extensiones de PHP**: Las siguientes extensiones son necesarias:
  - OpenSSL
  - PDO
  - Mbstring
  - Tokenizer
  - XML
  - Ctype
  - JSON
  - BCMath (para algunas características específicas)
  - Fileinfo
- **Base de Datos**: Laravel es compatible con varias bases de datos, entre las más comunes están:
  - MySQL
  - PostgreSQL
  - SQLite
  - SQL Server

1.4 Configuración de entorno

1.4.1 Instalación de Dependencias

- Instalar PHP y Composer:
  - En sistemas basados en Unix, puedes usar un gestor de paquetes como apt (para Ubuntu) o brew (para macOS).
  - En Windows, puedes usar instaladores como XAMPP, WAMP, LAMP para instalar PHP, y seguir las instrucciones de la web oficial de Composer.
  - Docker como creador de contenedores para el proyecto específico.

1.4.2 Instalar las extensiones de PHP

Asegúrate de que las extensiones mencionadas estén habilitadas en tu archivo php.ini.

1.4.3 Instalación y configuración

1.4.3.1 Instalar Laravel:

- Puedes crear un nuevo proyecto Laravel usando Composer. Ejecuta:

```
composer create-project --prefer-dist laravel/laravel nombre-del-proyecto "versión"
```

1.4.3.2 Configurar el servidor de desarrollo:

- Laravel viene con un servidor de desarrollo integrado que puedes iniciar con:

```
php artisan serve
```

- Esto ejecutará el servidor en `http://localhost:8000`.

1.4.3.2 Configurar la base de datos:

- Edita el archivo `.env` en la raíz de tu proyecto para configurar la conexión a tu base de datos.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=nombre_base_de_datos
DB_USERNAME=usuario
DB_PASSWORD=contraseña
```

1.5 Estructura de directorios

La estructura de directorios de Laravel está diseñada para ser intuitiva y organizada, facilitando la administración y escalabilidad del proyecto.

```
nombre-del-proyecto/
├── app/
│   ├── Console/
│   ├── Exceptions/
│   ├── Http/
│   │   ├── Controllers/
│   │   ├── Middleware/
│   ├── Models/
│   ├── Providers/
├── bootstrap/
│   ├── cache/
├── config/
├── database/
│   ├── factories/
│   ├── migrations/
│   ├── seeders/
├── lang/
├── public/
├── resources/
│   ├── js/
│   ├── lang/
│   ├── views/
│   ├── sass/
├── routes/
├── storage/
│   ├── app/
│   ├── framework/
│   │   ├── cache/
│   │   ├── sessions/
│   │   ├── views/
│   ├── logs/
├── tests/
│   ├── Feature/
│   ├── Unit/
├── vendor/
├── .env
├── .gitattributes
├── .gitignore
├── artisan
├── composer.json
├── composer.lock
├── package.json
├── phpunit.xml
├── README.md
├── webpack.mix.js
```

1.5.1 Descripción de los Directorios y Archivos Principales

1. **app/**:
   - **Console/**: Comandos de consola personalizados de Artisan.
   - **Exceptions/**: Manejo de excepciones personalizadas.
   - **Http/**:
   - **Controllers/**: Controladores de la aplicación.
   - **Middleware/**: Middleware HTTP para filtrar solicitudes.
   - **Models/**: Modelos Eloquent de la aplicación.
   - **Providers/**: Proveedores de servicios.
2. **bootstrap/**:
   - **cache/**: Archivo de caché generado para la configuración de la aplicación.
3. **config/**:
   - Contiene todos los archivos de configuración de la aplicación.
4. **database/**:
   - **factories/**: Factories para generar datos de prueba.
   - **migrations/**: Migraciones de la base de datos.
   - **seeders/**: Seeders para poblar la base de datos con datos iniciales.
5. **lang/**:
   - Archivos de localización para traducir la aplicación a diferentes idiomas.
6. **public/**:
   - Punto de entrada público de la aplicación. Aquí se encuentran index.php y los activos públicos como imágenes, JavaScript y CSS.
7. **resources/**:
   - **js/**: Archivos JavaScript/Vue.js de la aplicación.
   - **lang/**: Archivos de localización que se pueden utilizar en las vistas.
   - **views/**: Vistas Blade de la aplicación.
   - **sass/**: Archivos Sass/CSS de la aplicación.
8. **routes/**:
   - Archivos de rutas de la aplicación (web.php, api.php, console.php, channels.php).
9. **storage/**:
   - **app/**: Archivos de la aplicación.
   - **framework/**: Archivos generados por el framework como caché, sesiones y vistas compiladas.
   - **logs/**: Archivos de registro de la aplicación.
10. **tests/**:
    - **Feature/**: Pruebas funcionales.
    - **Unit/**: Pruebas unitarias.

11. **vendor/**:
    - Dependencias del proyecto instaladas por Composer.

1.5.2 Archivos Raíz

- **env**: Archivo de configuración de entorno.
- **.gitattributes** y **.gitignore**: Archivos de configuración de Git.
- **artisan**: Interfaz de línea de comandos de Laravel.
- **composer.json** y **composer.lock**: Archivos de configuración y bloqueo de dependencias de Composer.
- **package.json**: Archivo de configuración para npm/yarn.
- **phpunit.xml**: Configuración de PHPUnit.
- **README.md**: Archivo de documentación del proyecto.
- **webpack.mix.js**: Configuración de Laravel Mix para compilar activos

1.6 Configuración del entorno

1.6.1 Configuración de archivo (‘.env’)

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:...
APP_DEBUG=true
APP_URL=http://localhost

LOG_CHANNEL=stack

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=nombre_base_de_datos
DB_USERNAME=usuario
DB_PASSWORD=contraseña

BROADCAST_DRIVER=log
CACHE_DRIVER=file
FILESYSTEM_DRIVER=local
QUEUE_CONNECTION=sync
SESSION_DRIVER=file
SESSION_LIFETIME=120

MEMCACHED_HOST=127.0.0.1

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_MAILER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
MAIL_FROM_ADDRESS=null
MAIL_FROM_NAME="${APP_NAME}"

AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=
AWS_DEFAULT_REGION=us-east-1
AWS_BUCKET=

PUSHER_APP_ID=
PUSHER_APP_KEY=
PUSHER_APP_SECRET=
PUSHER_APP_CLUSTER=mt1
```

1.6.2 Claves de aplicación

Genera una clave de aplicación única, que se utiliza para cifrar datos sensibles. Ejecuta:

```
php artisan key:generate
```

1.6.3 Configuración de la base de datos

Edita la sección de la base de datos en el archivo .env para incluir los detalles de tu conexión de base de datos. Asegúrate de tener las credenciales correctas para DB_DATABASE, DB_USERNAME, y DB_PASSWORD.

1.6.4 Configuración del servidor de desarrollo

Para iniciar el servidor de desarrollo, ejecuta:

```
php artisan serve
```

Esto iniciará el servidor en http://localhost:8000 por defecto.

1.7 Rutas en Laravel

La configuración y uso de rutas en Laravel es fundamental para el manejo de solicitudes HTTP y la navegación en la aplicación. Las rutas definen cómo la aplicación debe responder a las diferentes solicitudes del navegador.

1.7.1 Archivo de rutas

Las rutas en Laravel se definen en archivos ubicados en el directorio routes/. Los archivos de rutas principales son:

- **web.php**: Rutas para la interfaz web que utilizan el middleware web.
- **api.php**: Rutas para la API que utilizan el middleware api.
- **console.php**: Definición de comandos de consola basados en Closure.
- **channels.php**: Rutas para canales de transmisión (Broadcasting).

1.7.2 Definición básica de rutas

En `routes/web.php`

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

// Ruta básica que devuelve una vista
Route::get('/', function () {
    return view('welcome');
});

// Ruta básica que devuelve una cadena de texto
Route::get('/hello', function () {
    return 'Hello, World!';
});
?>
```

1.7.3 Rutas con controladores

Laravel fomenta el uso de controladores para manejar la lógica de las rutas. Los controladores se encuentran en `app/Http/Controllers`.

Crear un Controlador

```
php artisan make:controller HomeController
```

Definir una Ruta que Utiliza un Controlador

```php
use App\Http\Controllers\HomeController;

Route::get('/home', [HomeController::class, 'index']);
```

Definición del Método en el Controlador

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class HomeController extends Controller
{
    public function index()
    {
        return view('home');
    }
}
```

1.7.4 Rutas con parámetros

Puedes definir rutas que acepten parámetros.

- Parámetros Obligatorios

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

- Parámetros Opcionales

```
Route::get('/user/{name?}', function ($name = 'Guest') {
    return 'User '.$name;
});
```

1.7.5 Rutas con nombres

Las rutas con nombre permiten generar URLs o redireccionar a rutas específicas fácilmente.

- Definir una Ruta con Nombre

```
Route::get('/profile', [UserController::class, 'show'])->name('profile');
```

- Generar URLs o Redireccionar Usando el Nombre de la Ruta

```
$url = route('profile');
```

```
return redirect()->route('profile');
```

1.7.6 Agrupación de rutas

Agrupar rutas permite compartir atributos como middleware o namespaces.

- Middleware

```
Route::middleware(['auth'])->group(function () {
    Route::get('/dashboard', function () {
        // Usará el middleware 'auth'
    });
});
```

- Namespaces

```
Route::namespace('Admin')->group(function () {
    // Controladores en el namespace App\Http\Controllers\Admin
    Route::get('/admin', 'DashboardController@index');
});
```

- Prefijos

```
Route::prefix('admin')->group(function () {
    Route::get('/users', function () {
        // Ruta es /admin/users
    });
});
```

1.7.7 Rutas API

Las rutas definidas en routes/api.php están pensadas para APIs y utilizan el middleware api.

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

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

1.7.8 Rutas de recursos

Las rutas de recursos son útiles para operaciones CRUD y siguen las convenciones RESTful.

```
Route::resource('photos', PhotoController::class);
```

Esto generará las siguientes rutas:

Método	Ruta	Acción	Nombre de la Ruta
GET	/photos	index	photos.index
GET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
GET	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update
DELETE	/photos/{photo}	destroy	photos.destroy

1.7.9 Rutas personalizadas

Puedes definir rutas personalizadas para manejar situaciones específicas.

- Rutas con expresiones regulares

```
Route::get('/user/{name}', function ($name) {
    return 'User '.$name;
})->where('name', '[A-Za-z]+');
```

- Subdominios

```
Route::domain('{account}.example.com')->group(function () {
    Route::get('user/{id}', function ($account, $id) {
        return 'User '.$id.' from '.$account;
    });
});
```

1.7.10 Middleware en rutas

El middleware puede ser aplicado a rutas específicas para realizar acciones antes o después de procesar la solicitud.

```
Route::get('/admin', function () {
    // Página de administración
})->middleware('auth');
```

1.7.11 Ejemplo de configuración de rutas

routes/web.php

```
use App\Http\Controllers\HomeController;
use Illuminate\Support\Facades\Route;

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

Route::get('/home', [HomeController::class, 'index'])->name('home');

Route::middleware(['auth'])->group(function () {
    Route::get('/dashboard', function () {
        return view('dashboard');
    })->name('dashboard');
});

Route::resource('photos', PhotoController::class);
```

Este ejemplo muestra cómo definir rutas básicas, utilizar controladores, aplicar middleware, nombrar rutas y definir rutas de recursos.

1.8 Controladores en Laravel

Los controladores en Laravel son clases que agrupan la lógica de manejo de solicitudes HTTP relacionadas. Cada método dentro de un controlador está diseñado para manejar una solicitud específica. Esto permite una organización más limpia y estructurada del código, en comparación con definir toda la lógica en las rutas.

1.8.1 Creación de controladores

Puedes crear un controlador utilizando el comando Artisan `make:controller`. Por ejemplo, para crear un controlador llamado `UserController`, usarías:

```
php artisan make:controller UserController
```

Esto generará un archivo en app/Http/Controllers llamado UserController.php.

1.8.2 Definición de métodos en el controlador

Un controlador típico puede tener varios métodos para manejar diferentes tipos de solicitudes HTTP. Por ejemplo:

```php
namespace App\Http\Controllers;

use Illuminate\Http\Request;


class UserController extends Controller
{
    // Método para manejar solicitudes GET a /users
    public function index()
    {
        // Lógica para obtener y devolver una lista de usuarios
    }

    // Método para manejar solicitudes POST a /users
    public function store(Request $request)
    {
        // Lógica para crear un nuevo usuario
    }

    // Método para manejar solicitudes GET a /users/{id}
    public function show($id)
    {
        // Lógica para obtener y devolver un usuario específico
    }

    // Método para manejar solicitudes PUT/PATCH a /users/{id}
    public function update(Request $request, $id)
    {
        // Lógica para actualizar un usuario específico
    }

    // Método para manejar solicitudes DELETE a /users/{id}
    public function destroy($id)
    {
        // Lógica para eliminar un usuario específico
    }
```

```
}
```

1.8.3 Rutas y controladores

Para asociar las rutas a los métodos del controlador, puedes definirlas en el archivo de rutas (`web.php` o `api.php`). Por ejemplo:

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

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

1.8.4 Controladores de recursos

Laravel también ofrece una manera rápida de crear controladores con un conjunto predefinido de métodos mediante el uso de controladores de recursos. Puedes crear un controlador de recursos usando:

```
php artisan make:controller UserController --resource
```

Y registrar las rutas automáticamente en `web.php` con:

```
Route::resource('users', UserController::class);
```

Esto creará todas las rutas RESTful necesarias para las operaciones CRUD.

1.8.5 Middlewares

Los controladores pueden usar middlewares para manejar tareas comunes como la autenticación y la autorización. Puedes aplicar un middleware a un controlador completo o a métodos específicos dentro de un controlador. Por ejemplo:

```
class UserController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth');
    }

    public function index()
    {
        // Este método solo es accesible por usuarios autenticados
    }
```

```
}
```

1.8.6 Inyección de dependencias

Decimos que se "inyecta una dependencia" cuando se pasa como argumento la instancia de una clase sobre otra, a través de uno de sus métodos. Laravel permite inyectar dependencias directamente en los métodos del controlador. Esto facilita la gestión de servicios y repositorios. Por ejemplo:

```php
use App\Services\UserService;

class UserController extends Controller
{
    protected $userService;

    public function __construct(UserService $userService)
    {
        $this->userService = $userService;
    }

    public function index()
    {
        $users = $this->userService->getAllUsers();
        return view('users.index', compact('users'));
    }
}
```

1.8.7 Validación de solicitudes

Laravel proporciona una manera conveniente de validar las solicitudes entrantes directamente en los controladores. Por ejemplo:

```php
use Illuminate\Http\Request;

class UserController extends Controller
{
    public function store(Request $request)
    {
        $validatedData = $request->validate([
            'name' => 'required|max:255',
            'email' => 'required|email|unique:users',
            'password' => 'required|min:8',
        ]);
```

```
        // Si la validación pasa, el código continúa aquí
    }
}
```

1.9 ¿Qué es Blade?

Blade es el motor de plantillas de Laravel que permite a los desarrolladores utilizar código PHP en las vistas de una manera estructurada y sencilla. A diferencia de otros motores de plantillas PHP, Blade no restringe al desarrollador a usar solo su propia sintaxis. Todo el código PHP nativo es perfectamente válido dentro de las vistas Blade.

1.9.1 Extensión de archivos Blade

Las vistas de Blade utilizan la extensión `.blade.php`. Por ejemplo, una vista puede llamarse `home.blade.php`.

1.9.2 Directivas de Blade

Blade ofrece una variedad de directivas que simplifican tareas comunes. Aquí hay algunas de las más utilizadas:

- **@extends**: Para extender una plantilla.
- **@section y @yield**: Para definir y renderear secciones de contenido.
- **@include**: Para incluir vistas.
- **@if, @elseif, @else y @endif**: Para condicionales.
- **@foreach, @for, @while y @endforeach, @endfor, @endwhile**: Para bucles.

1.9.3 Ejemplo básico

Supongamos que tienes una estructura básica de vistas en tu aplicación Laravel.
Layout principal (`layouts/app.blade.php`):

```
<!DOCTYPE html>
<html>
<head>
    <title>App Name - @yield('title')</title>
</head>
<body>
    <div class="container">
        @yield('content')
    </div>
</body>
</html>
```

Vista de contenido (`home.blade.php`):

```
@extends('layouts.app')

@section('title', 'Home Page')

@section('content')
    <h1>Welcome to the Home Page</h1>
    <p>This is the home page content.</p>
@endsection
```

1.9.4 Incluir otras vistas

Blade permite incluir otras vistas fácilmente con la directiva `@include`.

Vista principal (`layouts/app.blade.php`):

```
<!DOCTYPE html>
<html>
<head>
    <title>App Name - @yield('title')</title>
</head>
<body>
    @include('partials.header')

    <div class="container">
        @yield('content')
    </div>

    @include('partials.footer')
</body>
</html>
```

Vista de header (`partials/header.blade.php`):

```
<header>
    <h1>App Name</h1>
</header>
```

Vista de footer (`partials/footer.blade.php`):

```
<footer>
    <p>&copy; 2024 App Name</p>
</footer>
```

1.9.5 Mostrar datos y evitar XSS

* Cross-site scripting (XSS) es una vulnerabilidad de seguridad que permite a un atacante inyectar en un sitio web código malicioso del lado del cliente.

Blade utiliza la sintaxis de llaves dobles {{ }} para imprimir datos de una manera segura contra ataques XSS (Cross-Site Scripting). Para imprimir datos sin escaparlos, puedes usar {!! !!}.

```
<p>{{ $name }}</p> <!-- Escapa HTML -->
<p>{!! $name !!}</p> <!-- No escapa HTML -->
```

1.9.6 Componentes y slots

Blade también soporta componentes y slots, lo que permite crear fragmentos reutilizables de HTML con una sintaxis limpia.

Componente (`resources/views/components/alert.blade.php`):

```
<div class="alert alert-{{ $type }}">
    {{ $slot }}
</div>
```

Uso del componente:

```
<x-alert type="error">
    Something went wrong!
</x-alert>
```

1.9.7 Ciclo de vida de las vistas

Blade compila las vistas en código PHP plano y las almacena en caché para mejorar el rendimiento. Esto significa que las vistas solo se compilan una vez, a menos que se modifiquen.

1.9.8 Control de flujo y bucles

Blade proporciona directivas para manejar el control de flujo y los bucles de manera sencilla.

Condicionales

```
@if (count($records) === 1)
I have one record!
@elseif (count($records) > 1)
I have multiple records!
@else
I do not have any records!
```

```
@endif
```

Bucles

```
@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach
```

1.9.9 @csrf y @method

Al trabajar con formularios, Blade ofrece directivas para manejar tokens CSRF y métodos HTTP distintos a GET y POST.
Ejemplo de formulario:

```
<form action="/user" method="POST">
    @csrf
    @method('PUT')
    <input type="text" name="name" value="John Doe">
    <button type="submit">Submit</button>
</form>
```

1.9.10 Layouts

Un layout es una plantilla base que otras vistas pueden extender. Esto es útil para definir una estructura HTML común, como una cabecera, un pie de página y una sección de contenido.

```
<!DOCTYPE html>
<html>
<head>
    <title>App Name - @yield('title')</title>
    <link rel="stylesheet" href="{{ asset('css/app.css') }}">
</head>
<body>
    <header>
        @include('partials.header')
    </header>

    <div class="container">
        @yield('content')
    </div>

    <footer>
        @include('partials.footer')
```

```
    </footer>

    <script src="{{ asset('js/app.js') }}"></script>
</body>
</html>
```

1.9.10.1 Extender un Layout

Para utilizar un layout en una vista, utiliza la directiva `@extends` al principio del archivo de vista y define las secciones que deseas llenar con `@section`.

Ejemplo de una vista que extiende el layout **(`home.blade.php`)**:

```
@extends('layouts.app')

@section('title', 'Home Page')

@section('content')
    <h1>Welcome to the Home Page</h1>
    <p>This is the home page content.</p>
@endsection
```

1.9.10.2 Definir secciones

Las secciones se definen en el layout utilizando la directiva `@yield`. Puedes definir tantas secciones como necesites.
Ejemplo con más secciones:

```
<!DOCTYPE html>
<html>
<head>
    <title>App Name - @yield('title')</title>
    <link rel="stylesheet" href="{{ asset('css/app.css') }}">
    @yield('head')
</head>
<body>
    <header>
        @include('partials.header')
    </header>

    <div class="container">
        @yield('content')
    </div>
```

```
    <footer>
        @include('partials.footer')
    </footer>

    <script src="{{ asset('js/app.js') }}"></script>
    @yield('scripts')
</body>
</html>
```

Vista que llena las secciones adicionales (`home.blade.php`):

```
@extends('layouts.app')

@section('title', 'Home Page')

@section('head')
    <style>
        /* Custom styles for this page */
    </style>
@endsection

@section('content')
    <h1>Welcome to the Home Page</h1>
    <p>This is the home page content.</p>
@endsection

@section('scripts')
    <script>
        // Custom scripts for this page
    </script>
@endsection
```

1.9.10.3 Incluir vistas parciales

Para incluir fragmentos de vistas reutilizables, usa la directiva @include. Esto es útil para elementos como la cabecera, el pie de página, barras de navegación, etc.
Vista parcial para la cabecera (partials/header.blade.php):

```
<header>
    <h1>App Name</h1>
    <nav>
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/about">About</a></li>
```

```
            <li><a href="/contact">Contact</a></li>
        </ul>
    </nav>
</header>
```

1.10 Eloquent ORM

Eloquent ORM es una de las características más poderosas y populares de Laravel. Proporciona una forma intuitiva y sencilla de interactuar con la base de datos usando un enfoque orientado a objetos.

1.10.1 Configuración inicial

1.10.1.1 Configurar la Base de Datos

Configura tu conexión de base de datos en el archivo `.env` ubicado en la raíz de tu proyecto:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=nombre_bd
DB_USERNAME=usuario_bd
DB_PASSWORD=contraseña_bd
```

1.10.1.2 Creación de modelos y migraciones

Crear un Modelo y su Migración
Para crear un modelo y su migración correspondiente, usa el comando Artisan:

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

Esto creará un modelo en `app/Models/NombreModelo.php` y una migración en `database/migrations/timestamp_create_nombre_modelos_table.php`.

1.10.1.3 Definir la migración

En la migración, define la estructura de la tabla:

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

class CreateNombreModelosTable extends Migration
{
    public function up()
    {
        Schema::create('nombre_modelos', function (Blueprint $table) {
            $table->id();
            $table->string('nombre');
            $table->timestamps();
        });
    }
```

```
    public function down()
    {
        Schema::dropIfExists('nombre_modelos');
    }
}
```

1.10.1.4 Ejecutar migraciones

Para crear las tablas en la base de datos, ejecuta:

```
php artisan migrate
```

1.10.2 Uso de Eloquent ORM

Definir el Modelo: En el modelo `app/Models/NombreModelo.php`, define cualquier relación o configuración adicional:

```
namespace App\Models;

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

class NombreModelo extends Model
{
    use HasFactory;

    // Si el nombre de la tabla no sigue la convención de pluralización de
Laravel
    protected $table = 'nombre_modelos';

    // Si los timestamps no están en la tabla
    public $timestamps = false;

    // Especifica los campos que pueden ser llenados masivamente
    protected $fillable = ['nombre'];
}
```

1.10.3 Operaciones básicas con Eloquent

1.10.3.1 Crear un registro

```
use App\Models\NombreModelo;

$registro = NombreModelo::create(['nombre' => 'Ejemplo']);
```

1.10.3.2 Leer registros

```php
// Obtener todos los registros
$registros = NombreModelo::all();

// Obtener un registro por ID
$registro = NombreModelo::find(1);

// Obtener el primer registro que coincida con un criterio
$registro = NombreModelo::where('nombre', 'Ejemplo')->first();
```

1.10.3.3 Actualizar un registro

```php
$registro = NombreModelo::find(1);
$registro->nombre = 'Nuevo Nombre';
$registro->save();
```

1.10.3.4 Eliminar un registro

```php
//Eliminación física de un registro
$registro = NombreModelo::find(1);
$registro->delete();
```

1.10.4 Relaciones en Eloquent

1.10.4.1 Relación uno a muchos

```php
// En el modelo Padre (por ejemplo, Usuario)
public function posts()
{
    return $this->hasMany(Post::class);
}

// En el modelo Hijo (por ejemplo, Post)
public function user()
{
    return $this->belongsTo(User::class);
```

```
}
```

1.10.4.2 Relación muchos a muchos

```
// En ambos modelos (por ejemplo, Usuario y Rol)
public function roles()
{
    return $this->belongsToMany(Role::class);
}
```

1.10.5 Consultas avanzadas

1.10.5.1 Paginación

```
$registros = NombreModelo::paginate(15);
```

1.10.5.2 Consultas complejas

```
$registros = NombreModelo::where('nombre', 'like', '%ejemplo%')
->orderBy('created_at', 'desc')
->get();
```

1.10.6 Factories y Seeders

En el contexto de Laravel, tanto los Seeders como los Factories son herramientas que se utilizan para generar datos de prueba para una aplicación. Sin embargo, existen algunas diferencias clave entre ellos:

Propósito Los Seeders se utilizan para poblar la base de datos con datos de prueba, mientras que los Factories se utilizan para generar datos de prueba en tiempo de ejecución.

Tipo de datos: Los Seeders generan datos fijos y predefinidos, mientras que los Factories generan datos aleatorios y dinámicos.

Relación con los modelos: Los Seeders están asociados con los modelos de la aplicación y utilizan la sintaxis Eloquent para crear registros en la base de datos, mientras que los Factories están diseñados para generar datos independientes de los modelos.

Uso: Los Seeders se ejecutan una sola vez para poblar la base de datos con datos de prueba, mientras que los Factories se utilizan en pruebas automatizadas y en el desarrollo de características de la aplicación.
En resumen, los Seeders se utilizan para poblar la base de datos con datos fijos y predefinidos, mientras que los Factories se utilizan para generar datos aleatorios y dinámicos en tiempo de ejecución, y se utilizan principalmente en pruebas automatizadas y en el desarrollo de características de la aplicación.

1.10.6.1 Crear un Factory

```
php artisan make:factory NombreModeloFactory --model=NombreModelo
```

En `database/factories/NombreModeloFactory.php`:

```php
use App\Models\NombreModelo;
use Illuminate\Database\Eloquent\Factories\Factory;

class NombreModeloFactory extends Factory
{
    protected $model = NombreModelo::class;

    public function definition()
    {
        return [
            'nombre' => $this->faker->word,
        ];
    }
}
```

1.10.6.2 Usar Seeders

```
php artisan make:seeder NombreModeloSeeder
```

En `database/seeders/NombreModeloSeeder.php`:

```php
use Illuminate\Database\Seeder;
use App\Models\NombreModelo;

class NombreModeloSeeder extends Seeder
{
    public function run()
    {
        NombreModelo::factory()->count(50)->create();
    }
}
```

Para ejecutar seeders:

```
php artisan db:seed –class= NombreModeloSeeder
```

1.11 Migraciones

Las migraciones permiten definir y modificar la estructura de la base de datos usando código PHP en lugar de escribir directamente las consultas SQL. Esto facilita la colaboración entre desarrolladores y la gestión de cambios en la base de datos.

1.11.1 Creación de migraciones

1.11.1.1 Crear una Migración

Para crear una nueva migración, usa el comando `make:migration` de Artisan:

```
php artisan make:migration  create_nombretabla_table
```

Esto generará un archivo de migración en el directorio `database/migrations` con un nombre similar a `2023_06_22_000000_create_nombre_tabla_table.php`.

1.11.1.2 Definir la migración

En el archivo de migración, debes definir el esquema de la tabla en el método `up` y, opcionalmente, cómo revertir esos cambios en el método `down`.
Ejemplo de una migración básica para una tabla `posts`:

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

class CreatePostsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->text('body');
            $table->timestamps();
        });
    }
```

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

1.11.2 Ejecución de migraciones

1.11.2.3 Migrar la base de datos

Para aplicar todas las migraciones pendientes y crear las tablas correspondientes en la base de datos, usa el comando:

```
php artisan migrate
```

1.11.2.4 Revertir migraciones

Para revertir la última migración ejecutada, usa el comando:

```
php artisan migrate:rollback
```

Si necesitas revertir varias migraciones, puedes usar la opción `--step`:

```
php artisan migrate:rollback –step=3
```

Esto revertirá las últimas tres migraciones ejecutadas.

1.11.2.5 Refrescar migraciones

Para revertir y volver a ejecutar todas las migraciones, usa el comando:

```
php artisan migrate:refresh
```

Este comando es útil cuando deseas reconstruir la base de datos desde cero según las migraciones existentes.

1.11.2.6 Resetear migraciones

Para revertir todas las migraciones sin volver a ejecutarlas, usa el comando:

```
php artisan migrate:reset
```

1.11.2.7 Agregar columnas a una tabla con migración

Para añadir una columna a una tabla existente en Laravel debemos hacer uso de migraciones, para ello, primero debes crear una nueva migración usando el comando make:migration.

```
php artisan make:migration add_status_column_on_users_table --table=users
```

Donde:

- Add: operación.
- Status: atributo de la tabla.
- Users: nombre de la tabla.

1.11.2.8 Modificar un atributo a una tabla con migración

Para modificar una columna existente en una tabla en Laravel usando migraciones, primero debemos crear una nueva migración usando el comando make:migration.

```
php artisan make:migration modify_status_column_on_users_table --table=users
```

Donde:

- Add: operación.
- Status: atributo de la tabla.
- Users: nombre de la tabla.

1.11.2.9 Eliminar un atributo a una tabla con migración

El proceso para eliminar columnas utilizando migraciones en Laravel es el mismo, primero debemos crear una nueva migración usando el comando make:migration.

```
php artisan make:migration drop_ status column_on_users_table --table=users
```

1.11.3 Migraciones y seeders

Las migraciones a menudo se usan junto con los seeders para poblar la base de datos con datos iniciales.

Crear un Seeder, usa el comando:

```
php artisan make:seeder NombreTablaSeeder
```

Ejemplo de un seeder para la tabla `posts`:

```php
use Illuminate\Database\Seeder;
use App\Models\Post;

class PostsTableSeeder extends Seeder
{
    public function run()
    {
        Post::create([
            'title' => 'Primer Post',
```

```
            'body' => 'Contenido del primer post'
        ]);
    }
}
```

Para ejecutar un seeder, usa el comando:

```
php artisan db:seed –class=PostsTableSeeder
```

Para ejecutar todos los seeders registrados en `DatabaseSeeder`:

```
php artisan db:seed
```

1.11.4 Ejemplo de uso de migraciones

1. Crear una migración para la tabla posts

```
php artisan make:migration create_posts_table
```

2. Definir la estructura de la tabla en la migración

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

class CreatePostsTable extends Migration
{
    public function up()
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->text('body');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('posts');
    }
}
```

3. Ejecutar la migración

```
php artisan migrate
```

4. Crear un seeder para la tabla posts:

```
php artisan make:seeder PostsTableSeeder
```

5. Definir los datos de ejemplo en el seeder:

```
use Illuminate\Database\Seeder;
use App\Models\Post;
```

```
class PostsTableSeeder extends Seeder
{
    public function run()
    {
        Post::create([
            'title' => 'Primer Post',
            'body' => 'Contenido del primer post'
        ]);
    }
}
```

6. Ejecutar el seeder:

```
php artisan db:seed –class=PostsTableSeeder
```

1.12 Modelos

1.12.1 Crear un modelo

Para crear un modelo en Laravel, utilizamos el comando Artisan `make:model`. Este comando crea una clase de modelo en el directorio `app/Models`.

```
php artisan make:model NombreModelo
```

1.12.1.1 Crear un Modelo con Migración, Factory y Seeder

Puedes agregar opciones al comando `make:model` para crear automáticamente una migración, una factory y un seeder.

```
php artisan make:model NombreModelo -mfs
```

- `-m` crea una migración.
- `-f` crea una factory.
- `-s` crea un seeder.

1.12.2 Definir el modelo

El archivo del modelo se ubicará en `app/Models/NombreModelo.php`. Aquí puedes definir las propiedades y métodos del modelo.

```
namespace App\Models;

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

class NombreModelo extends Model
{
    use HasFactory;
```

```
    // Especificar la tabla si el nombre no sigue la convención pluralizada
    protected $table = 'nombre_modelos';

    // Especificar los campos que se pueden asignar masivamente
    protected $fillable = ['campo1', 'campo2'];

    // Ocultar campos en la serialización
    protected $hidden = ['campo_oculto'];

    // Definir relaciones, scopes, etc.
}
```

1.12.3 Definir relaciones

Eloquent facilita la definición de relaciones entre modelos.

1.12.3.1 Relación uno a muchos

En el modelo padre:

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

En el modelo hijo:

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

1.12.3.2 Relación muchos a muchos

En ambos modelos:

```
public function roles()
{
    return $this->belongsToMany(Rol::class);
}
```

1.12.4 Crear la migración

Si no creaste la migración junto con el modelo, puedes crearla manualmente:

```
php artisan make:migration create_nombremodelo _table
```

Define la estructura de la tabla en el archivo de migración:

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

class CreateNombreModelosTable extends Migration
{
    public function up()
    {
        Schema::create('nombre_modelos', function (Blueprint $table) {
            $table->id();
            $table->string('campo1');
            $table->string('campo2');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('nombre_modelos');
    }
}
```

Ejecuta la migración para crear la tabla:

```
php artisan migrate
```

1.12.5 Usa el modelo

Ahora que tienes el modelo y la tabla, puedes usar Eloquent para interactuar con la base de datos.

1.12.5.1 Crear un registro

```php
use App\Models\NombreModelo;

$registro = NombreModelo::create([
    'campo1' => 'Valor 1',
    'campo2' => 'Valor 2'
]);
```

1.12.5.2 Leer registros

```php
// Obtener todos los registros
$registros = NombreModelo::all();

// Obtener un registro por ID
$registro = NombreModelo::find(1);

// Obtener el primer registro que coincida con un criterio
$registro = NombreModelo::where('campo1', 'Valor 1')->first();
```

1.12.5.3 Actualizar un registro

```php
$registro = NombreModelo::find(1);
$registro->campo1 = 'Nuevo Valor 1';
$registro->save();
```

1.12.5.4 Eliminar un registro

```php
$registro = NombreModelo::find(1);
$registro->delete();
```

1.12.6 Ejemplo de uso de modelos

1. Crear el modelo con migración, factory y seeder

```
php artisan make:model Post -mfs
```

2. Definir el modelo ‘Post’

```php
namespace App\Models;

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

class Post extends Model
{
    use HasFactory;
```

```
    protected $fillable = ['title', 'body'];
}
```

3. Definir la migración

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

class CreatePostsTable extends Migration
{
    public function up()
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->text('body');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('posts');
    }
}
```

4. Ejecutar la migración

```
php artisan migrate
```

5. Crear y ejecutar un seeder (opcional)
    a. Definir el seeder

```
use Illuminate\Database\Seeder;
use App\Models\Post;

class PostsTableSeeder extends Seeder
{
    public function run()
    {
```

```
        Post::create([
            'title' => 'Primer Post',
            'body' => 'Contenido del primer post'
        ]);
    }
}
```

b. Ejecutar el seeder

```
php artisan db:seed –class=PostsTableSeeder
```

6. Usar el modelo 'Post' en el controlador o cualquier lugar de la aplicación

```
use App\Models\Post;

// Crear un post
$post = Post::create([
    'title' => 'Nuevo Post',
    'body' => 'Contenido del nuevo post'
]);

// Obtener todos los posts
$posts = Post::all();

// Actualizar un post
$post = Post::find(1);
$post->title = 'Título Actualizado';
$post->save();

// Eliminar un post
$post = Post::find(1);
$post->delete();
```

2 Practica 1

Genera un CRUD (Crear, Leer, Actualizar, Eliminar) en una entidad Post. Este ejercicio cubrirá la creación del modelo, migración, controlador, vistas y rutas necesarias.

2.1 Configuración del ambiente

2.1.2 Crear base de datos

Crea la base de datos:

```
CREATE DATABASE nombre_bd CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

2.1.2.1 Características de utf8mb4

Las mejores virtudes de UTF-8mb4 son:

- Soporte completo de Unicode, incluyendo emojis y caracteres especiales.
- Compatibilidad total con UTF-8, facilitando la migración y manteniendo la interoperabilidad.
- Internacionalización mejorada, ideal para aplicaciones globales con múltiples idiomas.
- Eficiencia en almacenamiento y transmisión debido a su codificación variable.
- Compatibilidad con tecnologías modernas y estándares como JSON y XML.

2.1.2.2 Características de utf8mb4_unicode_ci

UTF8MB4_unicode_ci es bueno porque:

- Ordenación y comparación precisas: Soporta el orden y comparación de texto basado en el estándar Unicode, lo cual es crucial para la correcta ordenación de múltiples idiomas.
- Compatibilidad con todos los caracteres Unicode, incluyendo aquellos fuera del plano básico multilingüe (BMP) como emojis.
- Manejo de acentos y diacríticos: Ignora diferencias entre acentos, haciendo las búsquedas más intuitivas.
- Internacionalización: Ideal para aplicaciones globales, facilitando la correcta gestión de datos en diversos idiomas.
- Eficiencia: Mantiene la eficiencia en operaciones de texto, incluso con la complejidad adicional de Unicode.

2.2 Configuración del proyecto

2.2.1 Crear un Nuevo Proyecto Laravel

Si no tienes un proyecto Laravel, crea uno nuevo, posiciónate dentro de la carpeta www del punto 2.1.1 para tu proyecto en Docker:

```
composer create-project --prefer-dist laravel/laravel blog "11.*"
```

2.2.2 Configura la base de datos

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=nombre_bd
DB_USERNAME=usuario_bd
DB_PASSWORD=contraseña_bd
```

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

2.3 Crear el modelo y la migración

2.3.1 Crear el Modelo `Post` con Migración

Utiliza Artisan para crear el modelo `Post` junto con su migración:

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

2.3.2 Definir la migración

Edita la migración generada en `database/migrations/xxxx_xx_xx_xxxxxx_create_posts_table.php`:

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

return new class extends Migration
{
    /**
     * Run the migrations.
     */
```

```
    public function up(): void
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->text('body');
            $table->timestamps();
        });
    }

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

2.3.3 Ejecutar la migración

Ejecuta las migraciones para crear la tabla en la base de datos:

```
php artisan migrate
```

2.4 Crear el controlador

2.4.1 Crear el Controlador `PostController`

Usa Artisan para crear un controlador de recursos:

```
php artisan make:controller PostController --resource
```

2.4.2 Definir Métodos del Controlador

Edita `app/Http/Controllers/PostController.php` para definir los métodos CRUD:

```
namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
```

```
     * Display a listing of the resource.
     */
    public function index()
    {
        $posts = Post::all();
        return view('posts.index', compact('posts'));
    }

    /**
     * Show the form for creating a new resource.
     */
    public function create()
    {
        return view('posts.create');
    }

    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request)
    {

        $validatedData = $request->validate([
            'title' => 'required|max:255',
            'body' => 'required'
        ]);
        Post::create($validatedData);

        return redirect()->route('posts.index')->with('success', 'Post
creado exitosamente');
    }

    /**
     * Display the specified resource.
     */
    public function show(string $id)
    {
        $post = Post::find($id);
        return view('posts.show',compact('post'));
    }

    /**
```

```
     * Show the form for editing the specified resource.
     */
    public function edit(string $id)
    {
        $post = Post::find($id);
        return view('posts.edit',compact('post'));
    }

    /**
     * Update the specified resource in storage.
     */
    public function update(Request $request, Post $post)
    {
        $validatedData = $request->validate([
            'title' => 'required|max:255',
            'body' => 'required',
        ]);

        $post->update($validatedData);

        return redirect()->route('posts.index')->with('success', 'Post
actualizado con éxito');
    }

    /**
     * Remove the specified resource from storage.
     */
    public function destroy(Post $post)
    {
        //SOLO PARA ELIMINACIONES FISICAS
        $post->delete();
        return redirect()->route('posts.index')->with('success','Post
eliminado exitosamente.');
    }
}
```

2.4.3 Definir datos para Modelo

Edita `app/Models/Post.php` para definir los datos de asignación masiva:

```
namespace App\Models;

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

```php
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use HasFactory;
    // Definir los campos que se permiten para asignación masiva
    protected $fillable = ['title', 'body'];
}
```

2.5 Definir rutas

Definir Rutas de Recursos

Edita `routes/web.php` para añadir las rutas de recursos para los posts:

```php
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\PostController;
Route::resource('posts',PostController::class);
```

2.6 Crear vistas

2.6.1 Crear la carpeta de vistas

En el directorio `resources/views`, crea una carpeta llamada `posts`.

2.6.2 Vista de inicio

Crea el archivo de plantilla `resources/views/posts/home.blade.php`:

```php
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Posts - @yield('title')</title>
</head>
<body>
    <h1>Posts</h1>
    <table class="table" border=1>
        <tr>
            <td style="width:50%;">
                <a href="{{ route('posts.index') }}">
                    Inicio
                </a>
```

```
            </td>
            <td style="width:50%;">
                <a href="{{ route('posts.create') }}">
                    Registro
                </a>
            </td>
        </tr>
    </table>
    <br><br>
    <div class="container">
        @yield('content')
    </div>
</body>
</html>
```

Crea el archivo `resources/views/posts/index.blade.php`:

```
@extends('posts.home')
@section('title','Inicio')
@section('content')
    <table class="table" border=1>
        @php
            $i=1;
        @endphp
        @foreach ($posts as $post)
            <tr>
                <td style="width: 5%">{{$i}}</td>
                <td style="width: 55%"><a href="{{ route('posts.show',
$post) }}">{{ $post->title }}</a></td>
                <td style="width: 20%"><a href="{{ route('posts.edit',
$post) }}">Editar</a></td>
                <td style="width: 20%">
                    <form action="{{ route('posts.destroy', $post) }}"
method="POST" style="display:inline;">
                        @csrf
                        @method('DELETE')
                        <button type="submit">Eliminar</button>
                    </form>
                </td>
            </tr>
            @php
```

```
                $i+=1;
            @endphp
        @endforeach
    </table>
@endsection
```

2.6.3 Vista de creación

Crea el archivo `resources/views/posts/create.blade.php`:

```
@extends('posts.home')
@section('title','Crear Post')
@section('content')
    <form action="{{ route('posts.store') }}" method="POST">
        @csrf
        <div>
            <label for="title">Título:</label>
            <input type="text" id="title" name="title" value="{{
old('title') }}">
                @error('title')
                    <div>{{ $message }}</div>
                @enderror
        </div>
        <br>
        <div>
            <label for="body">Contenido:</label>
            <textarea id="body" name="body">{{ old('body') }}</textarea>
                @error('body')
                    <div>{{ $message }}</div>
                @enderror
        </div>
        <br>
        <button type="submit">Crear</button>
    </form>
@endsection
```

2.6.4 Vista de edición

Crea el archivo `resources/views/posts/edit.blade.php`:

```
@extends('posts.home')
```

```
@section('title','Editar Post')
@section('content')
    <form action="{{ route('posts.update', $post) }}" method="POST">
        @csrf
        @method('PUT')
        <div>
            <label for="title">Título:</label>
            <input type="text" id="title" name="title" value="{{ $post-
>title }}">
                @error('title')
                    <div>{{ $message }}</div>
                @enderror
        </div>
        <br>
        <div>
            <label for="body">Contenido:</label>
            <textarea id="body" name="body">{{ $post->body }}</textarea>
                @error('body')
                    <div>{{ $message }}</div>
                @enderror
        </div>
        <br>
        <button type="submit">Actualizar</button>
    </form>
@endsection
```

2.6.5 Vista de mostrar

Crea el archivo `resources/views/posts/show.blade.php`:

```
@extends('posts.home')
@section('title',$post->title)
@section('content')
    <h1>{{ $post->title }}</h1>
    <p>{{ $post->body }}</p>
    <br>
    {{$post}}
    <br>
@endsection
```

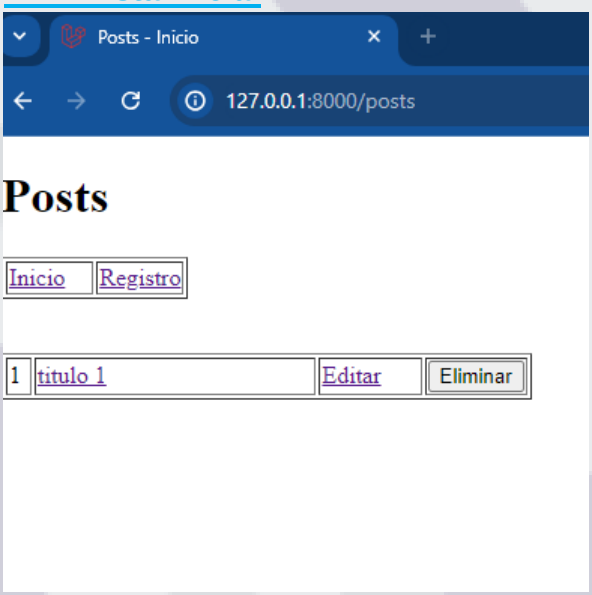
2.7 Prueba de aplicación

Inicia el servidor de desarrollo para probar tu aplicación:

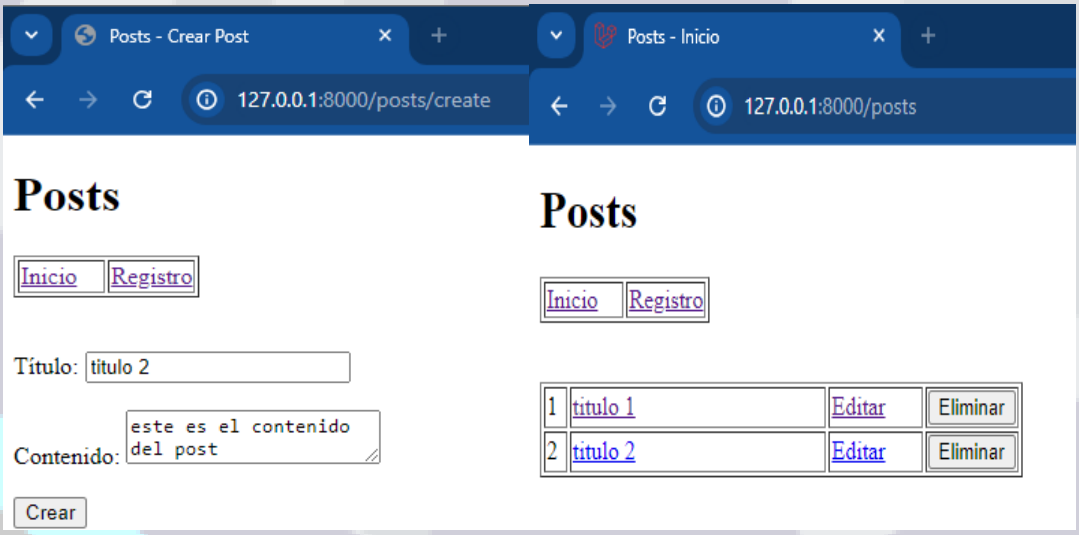
```
php artisan serve
```

Abre tu navegador y accede a `http://localhost:8000/posts`. Deberías ver la lista de posts y las opciones para crear, editar y eliminar posts.

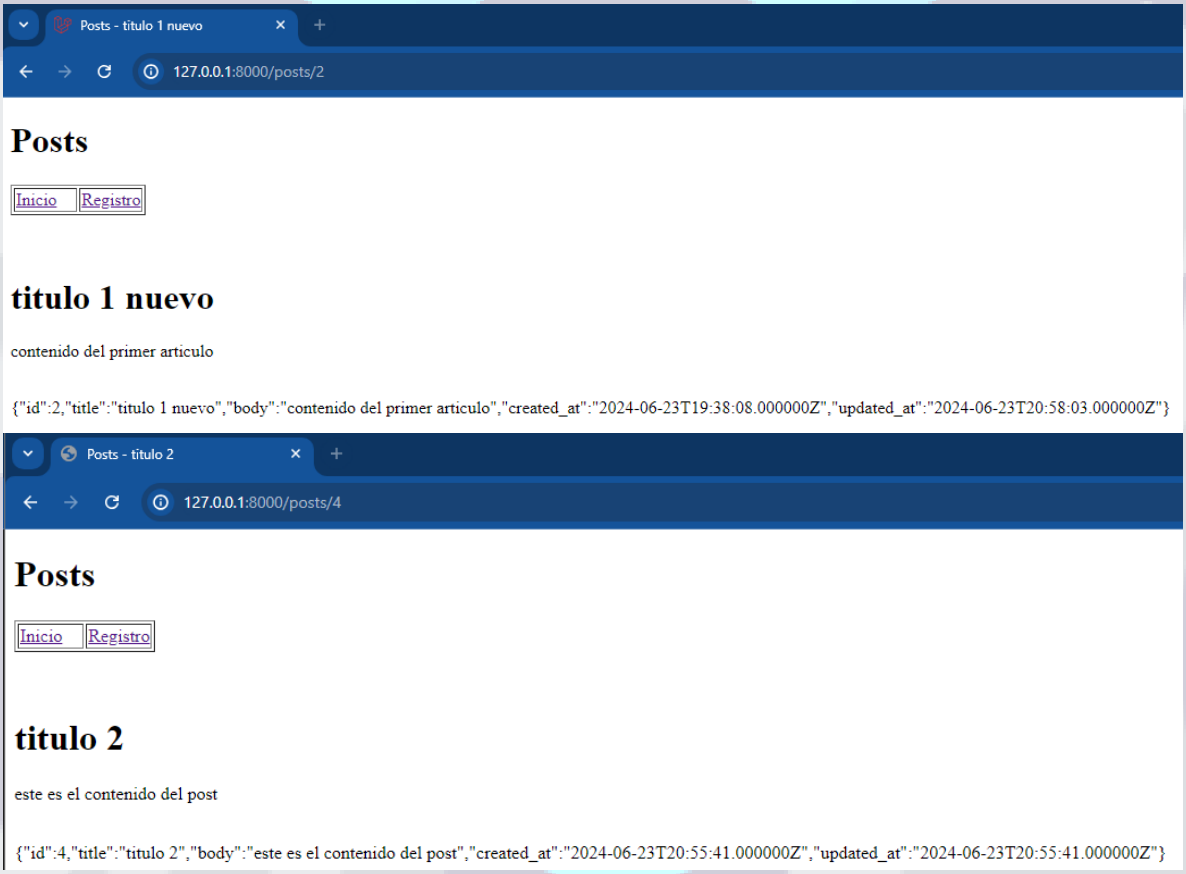
2.7.1 Vista inicial



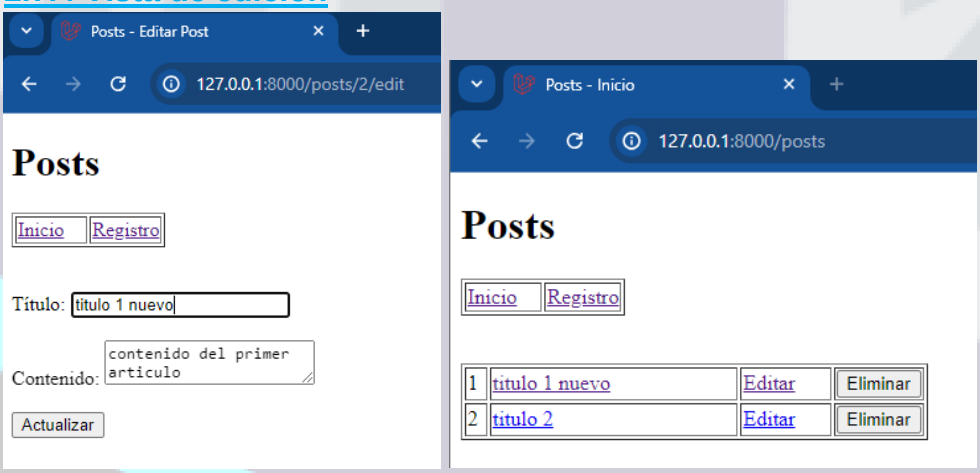
2.7.2 Vista de registro



2.7.3 Vista de consulta



2.7.4 Vista de edición



3 Autenticación

3.1 Instalación de Breeze

Instala el paquete de Breeze mediante Composer:

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

Ejecuta el instalador de Breeze:

```
php artisan breeze:install
```

*Nota: Breeze utiliza herramientas frontend como Vite. Asegúrate de tener Node.js y npm instalados. Luego, ejecuta:

```
npm install
npm run dev
```

Se requiere de migrar

```
php artisan migrate
```

O actualziar las migraciones (NOTA: este comando eliminara os datos y sobreescribira las migraciones)

```
php artisan refresh
```

3.2 Configuración de Rutas

Breeze configura automáticamente las rutas de autenticación. Puedes verlas en el archivo `routes/web.php`:

```php
use App\Http\Controllers\ProfileController;
use Illuminate\Support\Facades\Route;

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

Route::middleware('auth')->group(function () {
    Route::get('/dashboard', function () {
        return view('dashboard');
    })->name('dashboard');

    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');
    Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');
    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');
});
require __DIR__.'/auth.php';
```

3.3 Vistas de Autenticación

Las vistas de autenticación generadas por Breeze se encuentran en resources/views/auth. Estas vistas incluyen:

- login.blade.php
- register.blade.php
- forgot-password.blade.php
- reset-password.blade.php
- verify-email.blade.php
- confirm-password.blade.php

3.3.1 Personalización de Vistas

Puedes personalizar las vistas según tus necesidades. Por ejemplo, para personalizar la vista de inicio de sesión (`login.blade.php`), edita el archivo `resources/views/auth/login.blade.php`:

```
@extends('layouts.app')

@section('content')
<div class="container">
    <div class="row justify-content-center">
        <div class="col-md-8">
            <div class="card">
                <div class="card-header">{{ __('Login') }}</div>

                <div class="card-body">
                    <form method="POST" action="{{ route('login') }}">
                        @csrf

                        <div class="form-group row">
                            <label for="email" class="col-md-4 col-form-label text-md-right">{{
__('E-Mail Address') }}</label>

                            <div class="col-md-6">
                                <input id="email" type="email" class="form-control @error('email')
is-invalid @enderror" name="email" value="{{ old('email') }}" required autocomplete="email"
autofocus>

                                @error('email')
                                    <span class="invalid-feedback" role="alert">
                                        <strong>{{ $message }}</strong>
```

```
                                </span>
                            @enderror
                        </div>
                    </div>

                    <div class="form-group row">
                        <label for="password" class="col-md-4 col-form-label text-md-right">{{
__('Password') }}</label>

                        <div class="col-md-6">
                            <input id="password" type="password" class="form-control
@error('password') is-invalid @enderror" name="password" required autocomplete="current-password">

                            @error('password')
                                <span class="invalid-feedback" role="alert">
                                    <strong>{{ $message }}</strong>
                                </span>
                            @enderror
                        </div>
                    </div>

                    <div class="form-group row">
                        <div class="col-md-6 offset-md-4">
                            <div class="form-check">
                                <input class="form-check-input" type="checkbox" name="remember"
id="remember" {{ old('remember') ? 'checked' : '' }}>

                                <label class="form-check-label" for="remember">
                                    {{ __('Remember Me') }}
                                </label>
                            </div>
                        </div>
                    </div>

                    <div class="form-group row mb-0">
                        <div class="col-md-8 offset-md-4">
                            <button type="submit" class="btn btn-primary">
                                {{ __('Login') }}
                            </button>

                            @if (Route::has('password.request'))
                                <a class="btn btn-link" href="{{ route('password.request') }}">
```

```
                                    {{ __('Forgot Your Password?') }}
                                </a>
                            @endif
                        </div>
                    </div>
                </form>
            </div>
        </div>
    </div>
</div>
@endsection
```

3.4 Middleware de Autenticación

Para proteger rutas y permitir solo el acceso a usuarios autenticados, utiliza el middleware auth. Por ejemplo, para proteger la ruta del perfil:

```
Route::get('/profile', [ProfileController::class, 'edit'])->middleware('auth')->name('profile.edit');
```

3.5 Redirección Después de Login

Para personalizar la redirección después del login, edita el método `authenticated` en `app/Http/Controllers/Auth/LoginController.php`:

```
protected function authenticated(Request $request, $user)
{
    return redirect()->intended('/dashboard');
}
```

3.6 Verificación de Correo Electrónico

Para habilitar la verificación de correo electrónico, asegúrate de que el modelo User implemente MustVerifyEmail:

```
use Illuminate\Contracts\Auth\MustVerifyEmail;
class User extends Authenticatable implements MustVerifyEmail
{
    // ...
}
```

Esto habilitará automáticamente la verificación de correo electrónico y las rutas asociadas.

3.7 Modificacion de modelo User

Agrega el trait `HasRoles` al modelo `User` para habilitar la funcionalidad de roles y permisos:

```php
// app/Models/User.php
namespace App\Models;

use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Sanctum\HasApiTokens;
use Spatie\Permission\Traits\HasRoles;

class User extends Authenticatable /*implements MustVerifyEmail*/
{
    use HasApiTokens, HasFactory, Notifiable, HasRoles;

    // ...
}
```

*Si se desea una verificación por correo electrónico habrá que configurar el servidor SMTP y colcoar ‘implements MustVerifyEmail’ como parte de la definición de la clase User.

3.8 Roles y permisos

Ejecutrar en la terminal lo siguiente ya que instalar el paquete Spatie para ACL; de esa manera, podemos usar su método. Además, instalaremos el paquete de recopilación de formularios.

```
composer require spatie/laravel-permission
```

Recomendación: También podemos personalizar cambios en el paquete Spatie, por lo que si también desea realizar cambios, puede ejecutar el siguiente comando y obtener el archivo de configuración en config/permission.php y los archivos de migración.

```
php artisan vendor:publish --provider="Spatie\Permission\PermissionServiceProvider"
```

Se requiere de migrar

```
php artisan migrate
```

El paquete Spatie proporciona su middleware incorporado entonces, tenemos que agregar middleware en el archivo ‘\bootstrap\app.php’ de esta manera:

```php
use Illuminate\Foundation\Application;
use Illuminate\Foundation\Configuration\Exceptions;
use Illuminate\Foundation\Configuration\Middleware;
```

```
return Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        commands: __DIR__.'/../routes/console.php',
        health: '/up',
    )
    ->withMiddleware(function (Middleware $middleware) {
        $middleware->alias([
            'role' => \Spatie\Permission\Middleware\RoleMiddleware::class,
            'permission' => \Spatie\Permission\Middleware\PermissionMiddleware::class,
            'role_or_permission' =>
\Spatie\Permission\Middleware\RoleOrPermissionMiddleware::class
        ]);
    })
    ->withExceptions(function (Exceptions $exceptions) {
        //
    })->create();
```

A continuación se deberá de instalar el paquete laravel/ui como se muestra a continuación.

```
composer require spatie/laravel-permission
```

3.8.1 Creación de rol y permiso base

Puedes crear roles y permisos directamente en tu base de datos o mediante código en una seed. (database\seeders\RolesAndPermissionsSeeder.php)

```
php artisan make:seeder RolesAndPermissionsSeeder
```

Por ejemplo debería de verse de la siguiente manera:

```
namespace Database\Seeders;
use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Spatie\Permission\Models\Role;
use Spatie\Permission\Models\Permission;
class RolesAndPermissionsSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        // Crear permisos
```

```
        Permission::create(['name' => 'edit products']);
        Permission::create(['name' => 'delete products']);
        Permission::create(['name' => 'publish products']);
        Permission::create(['name' => 'unpublish products']);

        // Crear roles y asignar permisos
        $role = Role::create(['name' => 'almacen']);
        $role->givePermissionTo('edit products'); //Asignale el permiso edit a
rol almacen

        $role = Role::create(['name' => 'admin']);
        $role->givePermissionTo(Permission::all());
    }
}
```

Se ejecuta el seeder:

```
php artisan db:seed --class=RolesAndPermissionsSeeder
```

Esto anterior ejecutara las instrucciones que estén dentro del seeder.

3.8.2 Operaciones de asignación roles y usuarios

Para las diferentes operaciones entre permisos, roles y usuarios se identifican los siguientes fracmentos de código gracias al uso de *spatie*.

3.8.2.1 Creación de permisos

```
namespace Database\Seeders;

// use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Spatie\Permission\Models\Role;
use Spatie\Permission\Models\Permission;
use App\Models\User;

class RolesAndPermissionsSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        // Crear permisos
        Permission::create(['name' => 'edit products']);
        Permission::create(['name' => 'delete products']);
```

```
        Permission::create(['name' => 'publish products']);
        Permission::create(['name' => 'unpublish products']);
    }
}
```

3.8.2.2 Creacion de perfiles y asignación de los permisos a los mismos

```php
namespace Database\Seeders;

// use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Spatie\Permission\Models\Role;
use Spatie\Permission\Models\Permission;
use App\Models\User;

class RolesAndPermissionsSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        // Crear roles
        $roleAlmacen = Role::create(['name' => 'almacen']);
        $roleAdmin = Role::create(['name' => 'admin']);
        // Asignar permisos
        $roleAlmacen->givePermissionTo('edit products');
        $roleAdmin->givePermissionTo(Permission::all());//al rol admin
asignale todos los permisos
    }
}
```

3.8.2.3 Asignación, revocación y asignación masiva de permisos a un rol

```php
namespace Database\Seeders;

// use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Spatie\Permission\Models\Role;
use Spatie\Permission\Models\Permission;
use App\Models\User;

class RolesAndPermissionsSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        $roleAlmacen =Role::find(1); //Se busca el permiso 1
        $roleAdmin =Role::find(2); //Se busca el permiso 2
        //Asignacion de permisos
        Permission::find(1)->assignRole($roleAlmacen); //Al rol buscado con
id 1 asignale el permiso 1
        //Para revocar permisos
        Permission::find(1)->removeRole($roleAlmacen);
        //Para agregarles permisos de manera masiva a los roles; al rol 1 y
2 agregale el permiso 1
        Permission::find(1)->syncRoles(
            [
                $roleAlmacen,
                $roleAdmin
            ]
        );
    }
}
```

3.8.2.4 Asignación, revocación y asignación masiva de roles a usuarios

```php
namespace Database\Seeders;

// use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Spatie\Permission\Models\Role;
use Spatie\Permission\Models\Permission;
use App\Models\User;

class RolesAndPermissionsSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        $roleAlmacen =Role::find(1); //Se busca el permiso 1
        $roleAdmin =Role::find(2); //Se busca el permiso 2
        //Asignacion de roles a usuarios
        User::find(1)->assignRole($roleAdmin); //Agregale al usuario 1 el
rol 2
        User::find(1)->removeRole($roleAdmin); //Remover el rol 2 al
permiso 1
        User::find(1)->syncRoles([$roleAlmacen,$roleAdmin]); // asignarle
más de un rol a un usuario
    }
}
```

3.8.3 Validación de permisos en vistas (can)

Dentro de las vistas se pueden generar las directrices correspondientes a los permisos; para lo anterior se muestra el siguiente ejemplo de un menú y validación de permiso.

```
...
<!-- Navigation Links -->
                <div class="hidden space-x-8 sm:-my-px sm:ms-10 sm:flex">
                    <x-nav-link :href="route('dashboard')" :active="request()-
>routeIs('dashboard')">
                        {{ __('Inicio') }}
                    </x-nav-link>
                </div>
                <!-- Products Dropdown -->
                @can('read products')
                <div class="hidden space-x-8 sm:-my-px sm:ms-10  sm:items-center
sm:flex">
                    <x-dropdown align="right" width="48">
                        <x-slot name="trigger">
                            <button class="inline-flex items-center px-3 py-2 border
border-transparent text-sm leading-4 font-medium rounded-md text-gray-500 dark:text-
gray-400 bg-white dark:bg-gray-800 hover:text-gray-700 dark:hover:text-gray-300
focus:outline-none transition ease-in-out duration-150">
                                <div>Productos</div>
                                <div class="ms-1">
                                    <svg class="fill-current h-4 w-4"
xmlns="http://www.w3.org/2000/svg" viewBox="0 0 20 20">
                                        <path fill-rule="evenodd" d="M5.293 7.293a1 1 0
011.414 0L10 10.586l3.293-3.293a1 1 0 111.414 1.414l-4 4a1 1 0 01-1.414 0l-4-4a1 1 0
010-1.414z" clip-rule="evenodd" />
                                    </svg>
                                </div>
                            </button>
                        </x-slot>

                        <x-slot name="content">
                            <x-dropdown-link :href="route('products.index')">
                                Listado de productos
                            </x-dropdown-link>
                            @can('publish products')
                            <x-dropdown-link :href="route('products.create')">
                                Registro de productos
                            </x-dropdown-link>
```

```
                    @endcan
                </x-slot>
            </x-dropdown>
        </div>
        @endcan
```

…

En el anterior ejemplo se coloca la directiva can (@can('read products')), en esta se engloban los datos que serán validados, por ejemplo aquí se verifica que en la tabla de 'role_has_permissions' se encuentre el permiso al rol del usuario loegado si cuenta con el permiso podrá ver la sección del menú en caso contrario no lo vera.
El comando inicia con un @can('nombre_de_permiso') y termina en un @endcan.

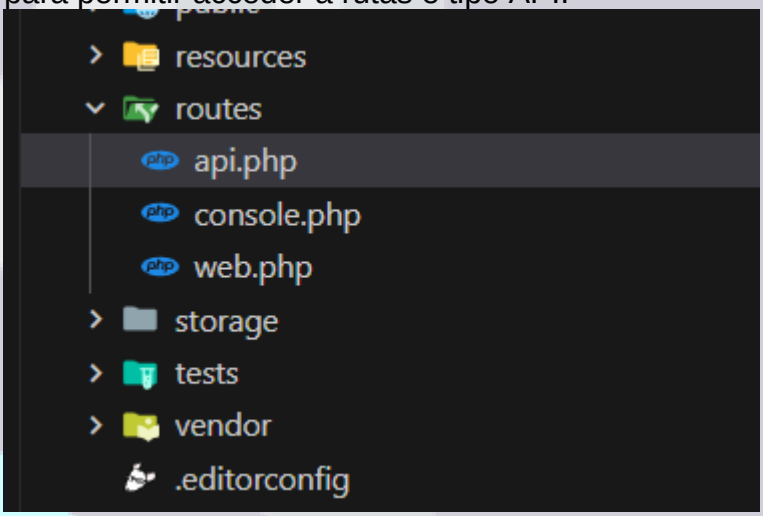
4 API Resources

4.1 Requerimientos para creación

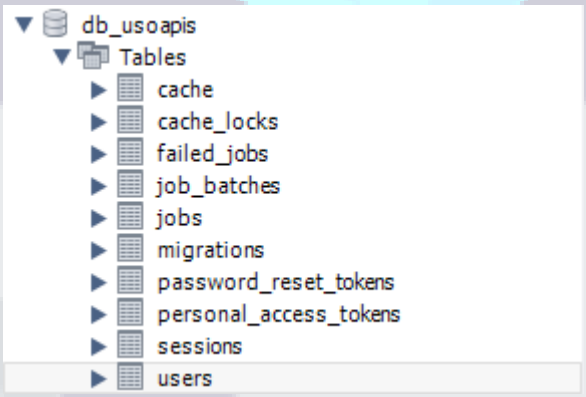
Ejecuta el comando:

```
php artisan install:api
```

Este comando nos generara el archivo de rutas y las configuraciones necesarias para permitir acceder a rutas e tipo API.



De la misma manera se generara una migración y algunas tablas dentro de la base de datos.



4.2 Declaración de rutas API

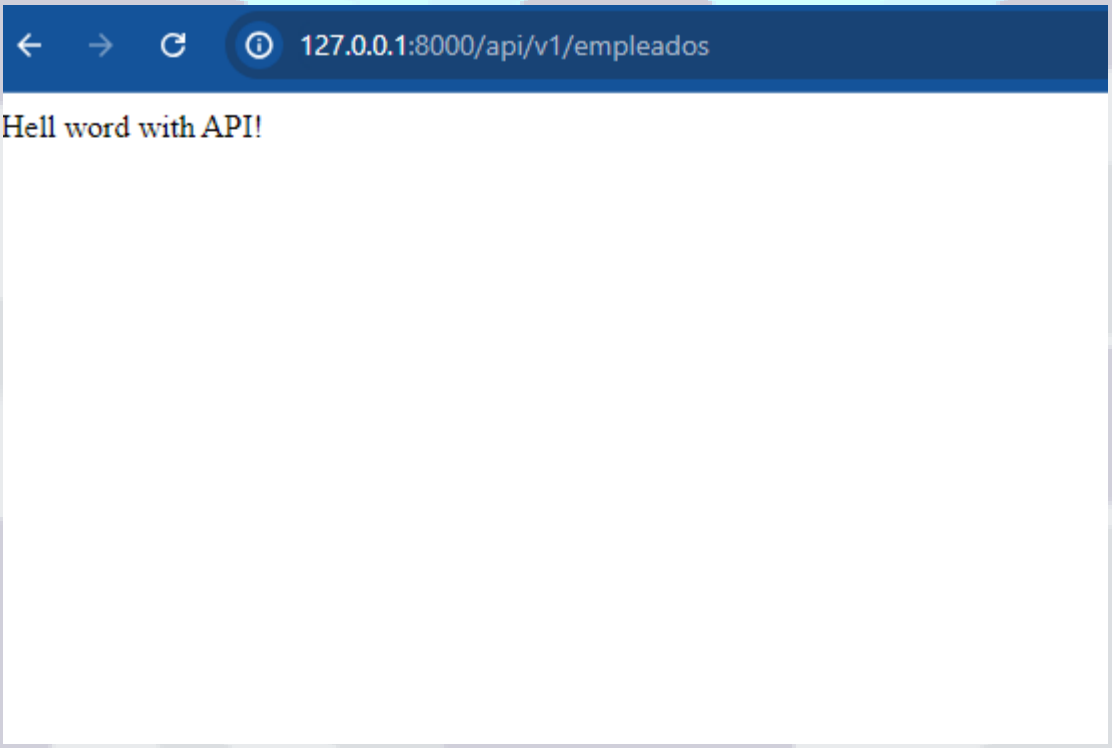
La declaración de rutas es similar a las declaradas en el archivo web.php por ejemplo por defecto se crea una para usuarios como lo siguiente:

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

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

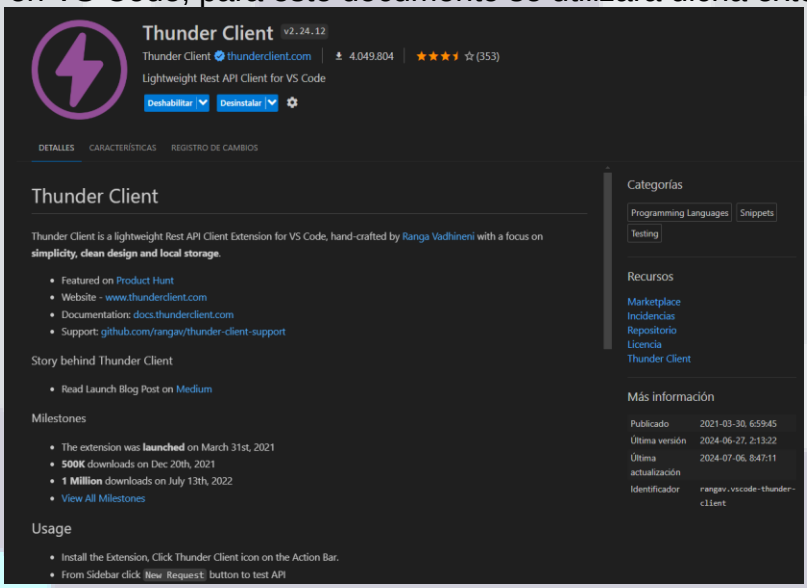
Para hacer un ejemplo más practivo le llamaremos empleados a una nueva ruta.

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

Route::prefix('v1')->group(function () {
    Route::get('/empleados', function () {
        return "Hell word with API!";
    });
});
```

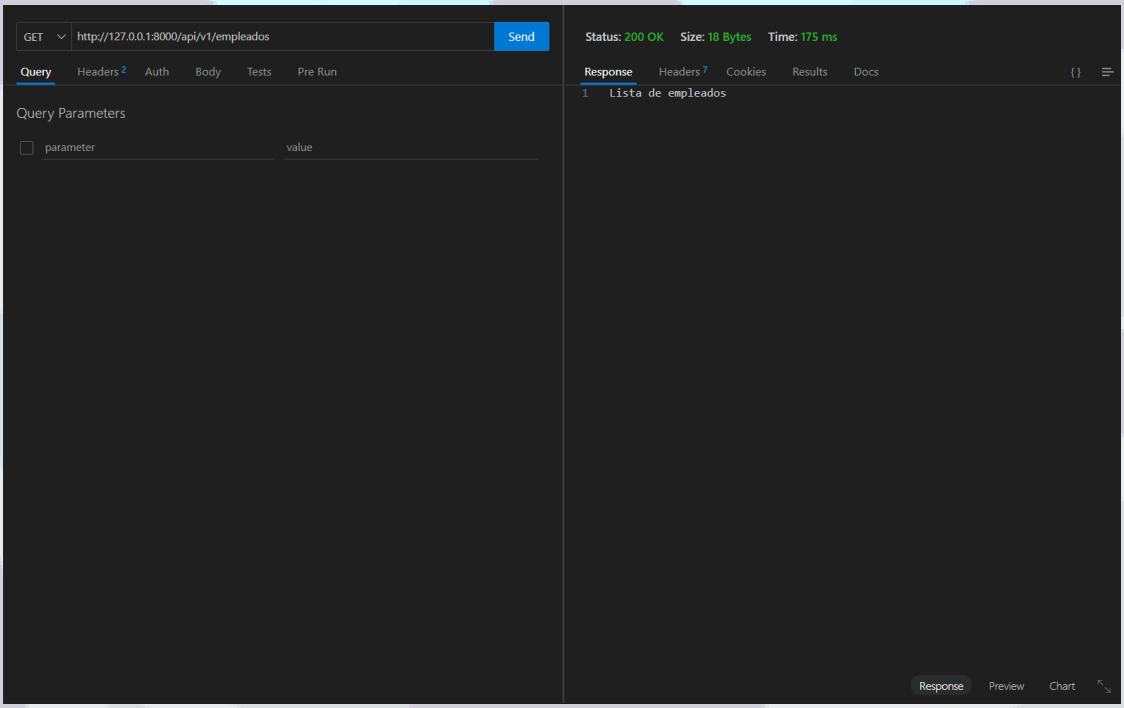
Lo anterior al ejecutar el servidor se podrá ver de la siguiente manera:



NOTA: dentro de las extensiones de VS Code se puede hacer uso de una extensión de nombre ThunderClient la cual permite realziar pruebas de API directamente en VS Code, para este documento se utilizara dicha extensión.



La anterior prueba realizada en el navegador con ThunderClient se ve de la siguiente manera:



Para ejemplificar mejor se realziara la creación de una nuevamigración que almacenara los datos de los empleados, para ello ejecutaremos el comando:

```
php artisan make:migration  create_employed_table
```

Con esto crearemos la migración y los datos que almacenaremos dentro de employed, al abrir el archivo de la migración en *'uso_apis\database\migrations\2024_07_06_145546_create_employed_table.php'* se puede ver el contenido base, el cual modificaremos para que quede de la siguiente manera:

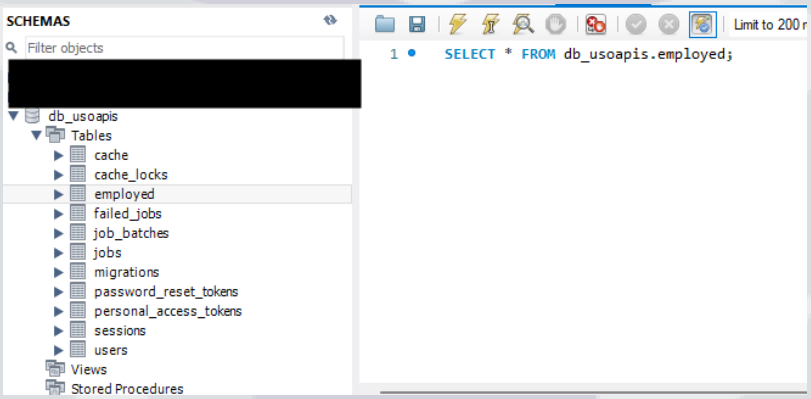
```php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('employed', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email');
            $table->string('phone');
            $table->string('area');
```

```
            $table->timestamps();
        });
    }

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

Ahora se ejecutara la migración para poder crear la tabla:

```
php artisan migrate
```

Y con esto estará creada la tabla en la base de datos:



Ahora para poder interactuar con la base se deberá de crear un modelo con el nombre de la tabla que se creo para lo que en este ejemplo seria el siguiente:

```
php artisan make:model Employed
```

Esto nos creara la estructura del modelo en la ruta ‘uso_apis\app\Models\Employed.php’ por defecto nos contiene lo siguiente:

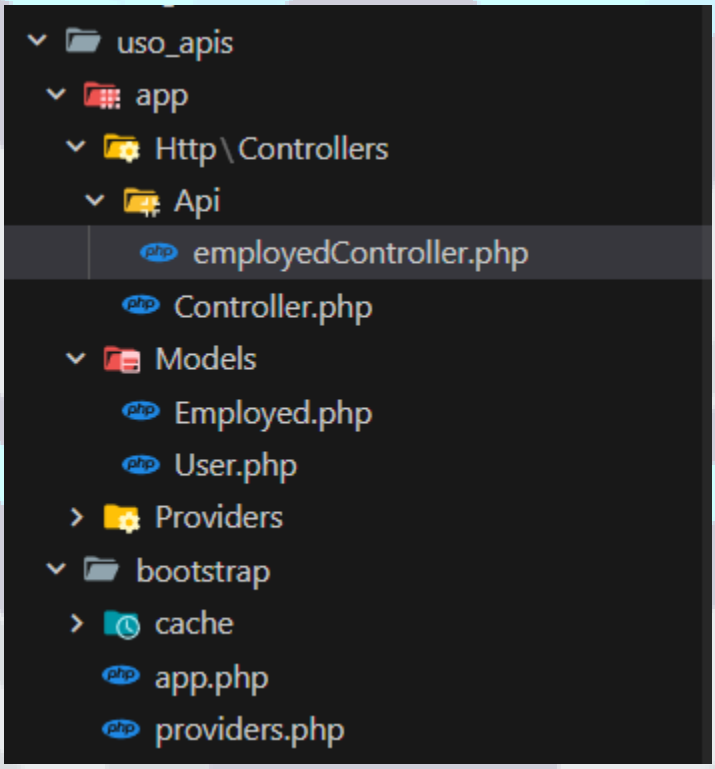
```
namespace App\Models;

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

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

Se deberá de tropicalizar para poder intercatuar con lso atributos de la migración (tabla de la base de datos),

```php
namespace App\Models;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
class Employed extends Model
{
    use HasFactory;
    //Nombre de la tabla creada en la base de datos (nombre de la migracion)
    protected $table='employed';
    //Datos que permitire actualizar
    protected $fillable =[
        'name',
        'email',
        'phone',
        'area'
    ];
}
```

De la misma manera se deberá de crear un controller que nos permita interactuar con el modelo siguiento el patrón de diseño MVC, como el API en este caso requiere realizar todas las operaciónes del CRUD se coloca la instrucción '--resource'que nos generara todas las operaciones dentro del controller (para el caso de este controller y para un mejor manejo de archivos se creara dentro de la carpeta API), por lo que se ejecutara el comando:

```
php artisan make:controller Api/employedController --resource
```



4.3 Operaciones con API

Dentro del controller se realizaran las operaciones necesarias para que la API por medio de la ruta pueda generar las operaciónes.

4.3.1 Consulta listado de la tabla

Dentro del controller (uso_apis\app\Http\Controllers\Api\employedController.php) se coloca algo como lo siguiente en el método index:

```php
namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Models\Employed;

class employedController extends Controller
{
    /**
     * Display a listing of the resource.
     */
    public function index()
```

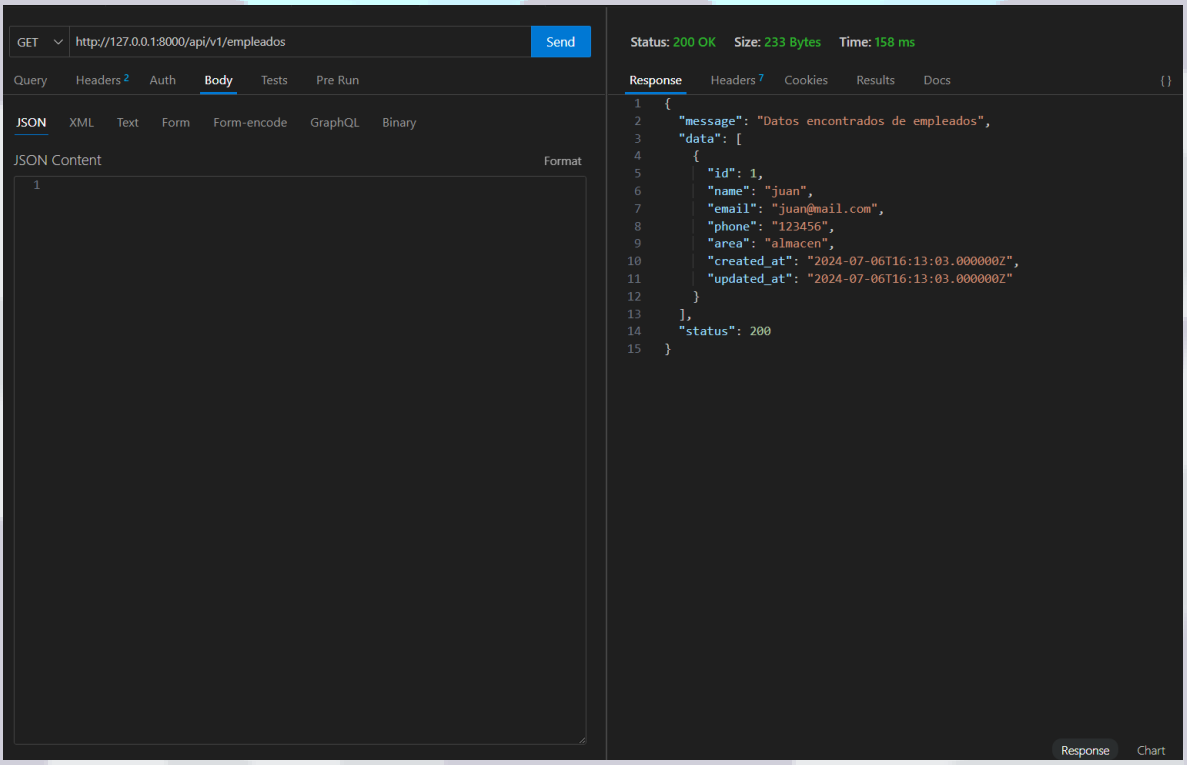
```
    {
        $employees=Employed::all();
        $data=[
            'message'=>'Datos encontrados de empleados',
            'data'=>$employees,
            'status'=>200
        ];
        if($employees->isEmpty()){
            $data['message']='No se encontraron empleados';
        }
        return response()->json($data,200);
    }
}
```

NOTA: recordar que se debe de importar al modelo para poder hacer el llamado a sus datos (use App\Models\Employed;).

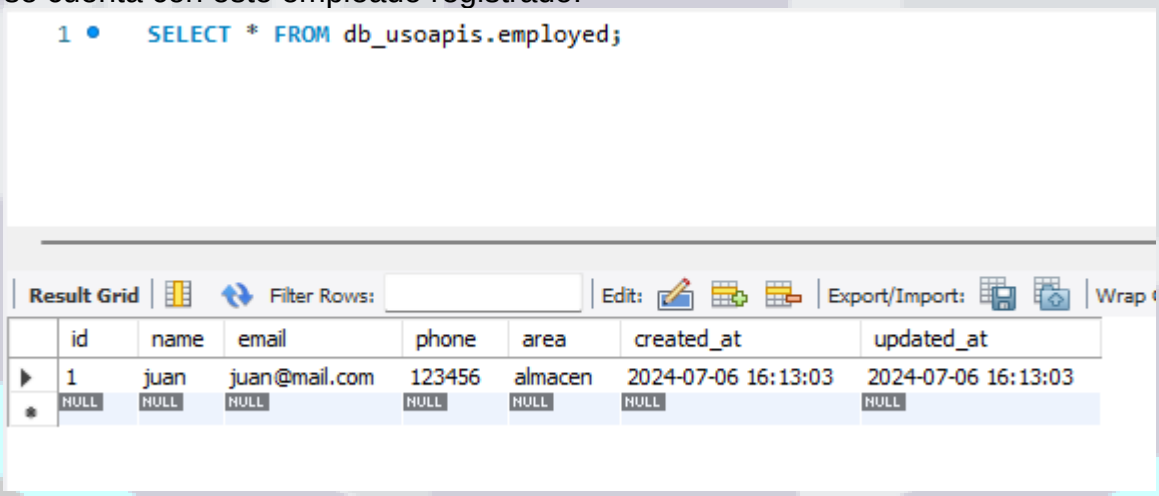
Dentro de la ruta del API se deberá de definir la ruta al método index como se muestra a continuación:

```
use App\Http\Controllers\Api\employedController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;
Route::prefix('v1')->group(function () {
    Route::get('/empleados', [employedController::class,'index']);

    Route::get('/empleados/{id}', function () { return "Ver un empleado"; });
    Route::post('/empleados', function () { return "Creacion de empleados"; });
    Route::put('/empleados/{id}', function () { return "Actualizando empleados"; });
    Route::put('/empleados/{id}', function () { return "Eliminando empleados"; });
});
```

Al realizar la prueba con ThunderClient se puede validar lo siguiente:



Esto nos indica que en la tabla efectivamente se cuenta con un dato, si consultamos la tabla desde un gestor de bases podríamos ver que efectivamente se cuenta con este empleado registrado:



4.3.2 Creación de registros

Dentro del controller se deberá de colocar lo siguiente en el método store:

```php
namespace App\Http\Controllers\Api;
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Models\Employed;
use Illuminate\Support\Facades\Validator;

class employedController extends Controller
{
    /**
     * Display a listing of the resource.
     */
    public function index()
    {…}
    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(),[
            'name'=>'required',
            'email'=>'required|email',
            'phone'=>'required',
            'area'=>'required',
        ]);
        $data=[
            'message'=>'Error al ejecutar creación de empleado',
            'data'=>$validator->errors(),
            'status'=>400
        ];
        if($validator->fails()){
            $data=[
                'message'=>'Error en validacion de datos',
                'data'=>$validator->errors(),
                'status'=>400
            ];
        }else{
            $employed = Employed::create([
                'name'=>$request->name,
                'email'=>$request->email,
                'phone'=>$request->phone,
```

```
                'area'=>$request->area,
            ]);
            if(!$employed){
                $data=[
                    'message'=>'Error al crear empleado',
                    'status'=>500
                ];
            }else{
                $data=[
                    'message'=>'Estudiante creado.',
                    'data'=>$employed,
                    'status'=>200
                ];
            }
        }
        return response()->json($data);
    }
}
```

Para realizar la validación de formulario en esta ocasión usaremos el modelo 'Validator' el cual deberemos definir en la parte superior (use Illuminate\Support\Facades\Validator;), primero se valida la información que queremos validar (más información aqui):

```
$validator = Validator::make($request->all(),[
            'name'=>'required',
            'email'=>'required|email',
            'phone'=>'required',
            'area'=>'required',
        ]);
```

Algunas validaciones más específicas o avanzadas podrían ser:

```
$validator = Validator::make($request->all(),[
            'name'=>'required|max:255',
            'email'=>'required|email|unique:employed',
            'phone'=>'required|digits:10',
            'area'=>'required|in:Almacen,Ventas,Administrativo',
        ]);
```

Una vez validado se ejecuta la creación o guardado de datos.

```
$employed = Employed::create([
                'name'=>$request->name,
                'email'=>$request->email,
                'phone'=>$request->phone,
                'area'=>$request->area,
            ]);
```

4.3.3 Consulta de un registro único

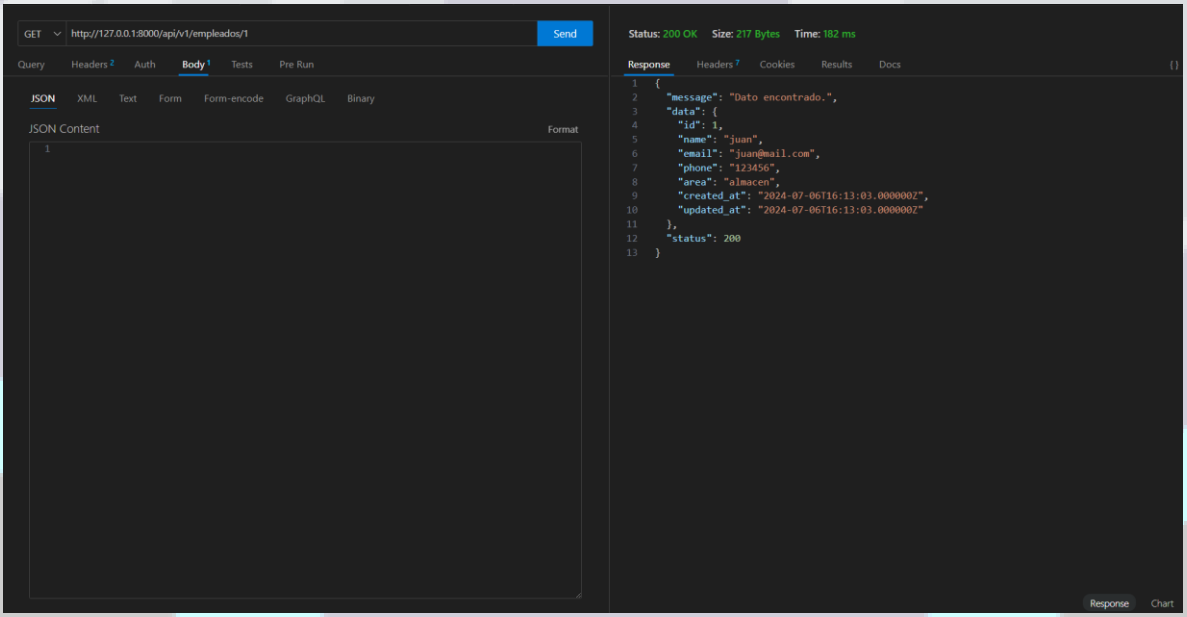
Para realizar consultas de registros únicos por medio del id se requiere colorar dicho método dentro del controller:

```
namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Models\Employed;
use Illuminate\Support\Facades\Validator;

class employedController extends Controller
{
    public function index()
    {…}
    public function store(Request $request)
    {…}
    public function show(string $id)
    {
        $employed = Employed::find($id);
        if(!$employed){
            $data=['message'=>'Error empleado no encontrado',
'data'=>array(),'status'=>404];
        }else{
            $data=['message'=>'Dato encontrado.', 'data'=>$employed, =>200];
        }
        return response()->json($data);
    }
}
```

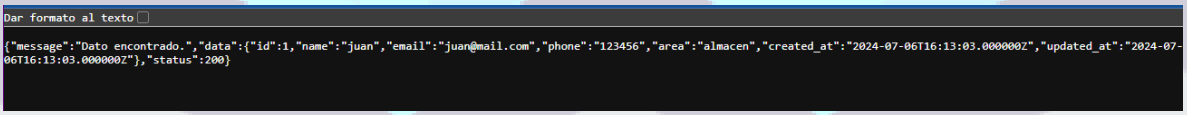
Dentro de las rutas al igual que en los anteriores casos se deberá de colocar la que haga alusión al método requerido y pasar por parámetro el identificador a buscar:

```php
use App\Http\Controllers\Api\employedController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;
/* Route::get('/user', function (Request $request) { return $request->user(); })->middleware('auth:sanctum'); */
Route::prefix('v1')->group(function () {
    Route::get('/empleados', [employedController::class,'index']);
    Route::post('/empleados', [employedController::class,'store']);
    Route::get('/empleados/{id}', [employedController::class,'show']);

    Route::put('/empleados/{id}', function () { return "Actualizando empleados"; });
    Route::put('/empleados/{id}', function () { return "Eliminando empleados"; });
});
```

Al realziar las pruebas se pueden constatar por medio del navegador o por ThunderClient, para ejemplificar realizaremos ambos:
ThunderClient:



Navegador:



4.3.4 Eliminar un registro

Al igual que buscar un registro se realiza la búsqueda del registro a borrar y se ejecuta lo necesario tal y como se muestra en el método a continuación:

```php
namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Models\Employed;
use Illuminate\Support\Facades\Validator;

class employedController extends Controller
{
    public function index()
    {…}
    public function store(Request $request)
    {…}
    public function show(string $id)
    {…}
    public function destroy(string $id)
    {
        $employed = Employed::find($id);
        if(!$employed){
            $data=[
                'message'=>'Error empleado no encontrado',
                'data'=>array(),
                'status'=>404
            ];
        }else{
            $employed->delete();
            $data=[
                'message'=>'Registro de empleado eliminado.',
                'data'=>$employed,
                'status'=>200
            ];
        }
        return response()->json($data);
```

```
    }
```

Para el caso de las rutas es similar a las anteriores, pero en necesario tomar en cuenta que en el method de la ruta debe de ponerse ‘delete’:

```
use App\Http\Controllers\Api\employedController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;
Route::prefix('v1')->group(function () {
    Route::get('/empleados', [employedController::class,'index']);
    Route::post('/empleados', [employedController::class,'store']);
    Route::get('/empleados/{id}', [employedController::class,'show']);
    Route::delete('/empleados/{id}', [employedController::class,'destroy']);

    Route::put('/empleados/{id}', function () { return "Actualizando
empleados"; });
});
```

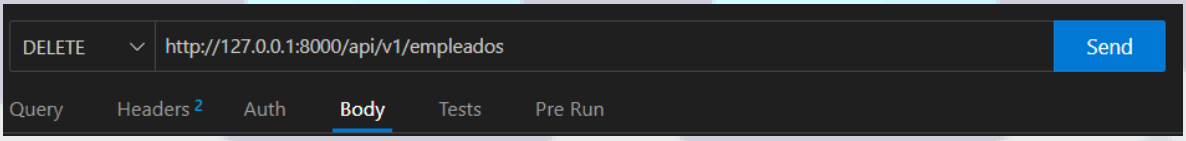
A continuación se muestra el listado de usuarios con los que se cuentan:

Status: 200 OK Size: 410 Bytes Time: 6.54 s

Response Headers 7 Cookies Results Docs

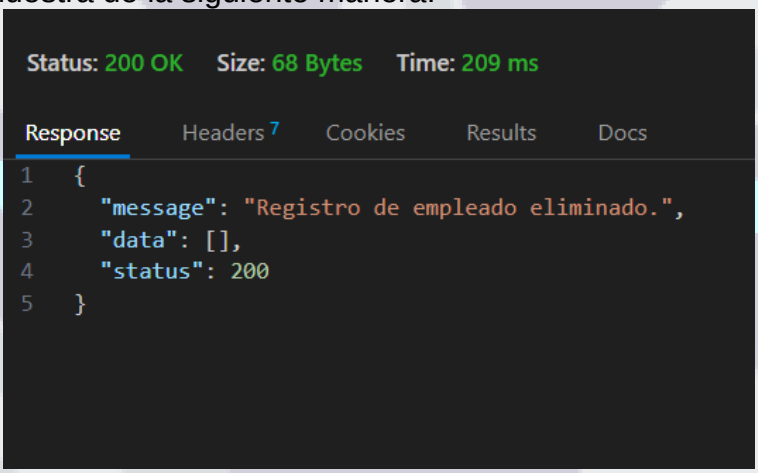

```
{
  "message": "Datos encontrados de empleados",
  "data": [
    {
      "id": 1,
      "name": "juan",
      "email": "juan@mail.com",
      "phone": "123456",
      "area": "almacen",
      "created_at": "2024-07-06T16:13:03.000000Z",
      "updated_at": "2024-07-06T16:13:03.000000Z"
    },
    {
      "id": 2,
      "name": "juanito",
      "email": "juanito@mail.com",
      "phone": "1234567890",
      "area": "Almacen",
      "created_at": "2024-07-06T16:49:10.000000Z",
      "updated_at": "2024-07-06T16:49:10.000000Z"
    }
  ],
  "status": 200
}
```


Para este ejercicio se eliminara el usuario con id 1 por lo que dentro de ThunderClient se coloca en la URL lo siguiente:

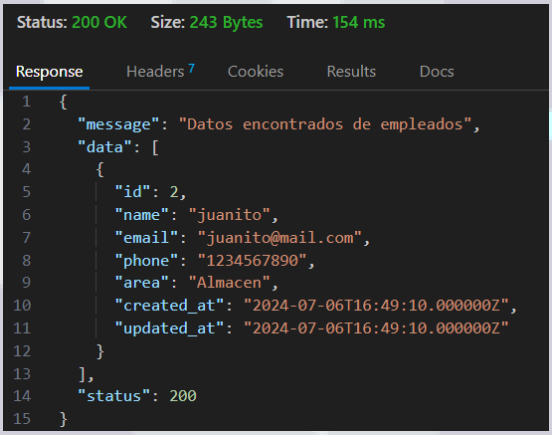


Recordando que se debe de seleccionar el method ‘DELETE’.

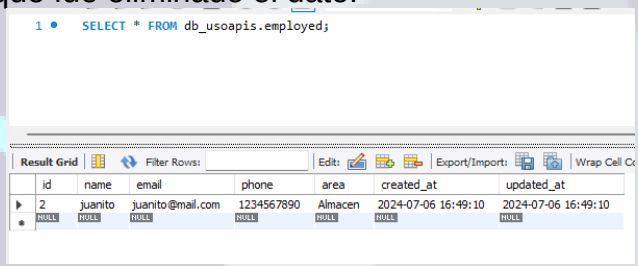
Al ejecutar nos mostrara el mensaje que prefeinimos dentro del controller, para este caso se muestra de la siguiente manera:



Si realizamos la consulta nos muestra que fue eliminado el dato:



Vía API



Vía Base de datos

4.3.4 Actualizar un registro

4.3.4.1 Ejemplo con PUT

La actualización es una combinación de los anteriores métodos para lo que definimos lo siguiente:

```
namespace App\Http\Controllers\Api;
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Models\Employed;
use Illuminate\Support\Facades\Validator;

class employedController extends Controller
{
    public function index()
    {…}
    public function store(Request $request)
    {…}
    public function show(string $id)
    {…}
    public function destroy(string $id)
    {…}
    public function update(Request $request, string $id)
    {
        $employed = Employed::find($id);
        if(!$employed){
            $data=[
                'message'=>'Error empleado no encontrado',
                'data'=>array(),
                'status'=>404
            ];
        }else{
            $validator = Validator::make($request->all(),[
                'name'=>'required|max:255',
                'email'=>'required|email|unique:employed',
                'phone'=>'required|digits:10',
                'area'=>'required|in:Almacen,Ventas,Administrativo',
            ]);
            $data=[
                'message'=>'Error al ejecutar creación de empleado',
                'data'=>$validator->errors(),
                'status'=>400
            ];
```

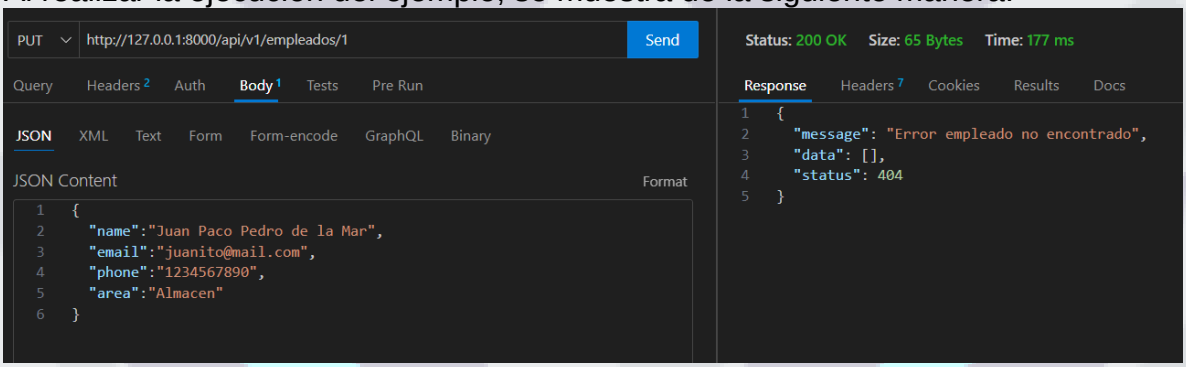
```
            if($validator->fails()){
                $data=[
                    'message'=>'Error en validacion de datos',
                    'data'=>$validator->errors(),
                    'status'=>400
                ];
            }else{
                $employed->name=$request->name;
                $employed->email=$request->email;
                $employed->phone=$request->phone;
                $employed->area=$request->area;

                $data=[
                    'message'=>'Empleado actualizado.',
                    'data'=>$employed,
                    'status'=>200
                ];
            }
        }
        return response()->json($data);
    }
}
```

La definición de las rutas para el method ‘PUT’ que corresponde a la actualziacion queda de la siguiente manera:

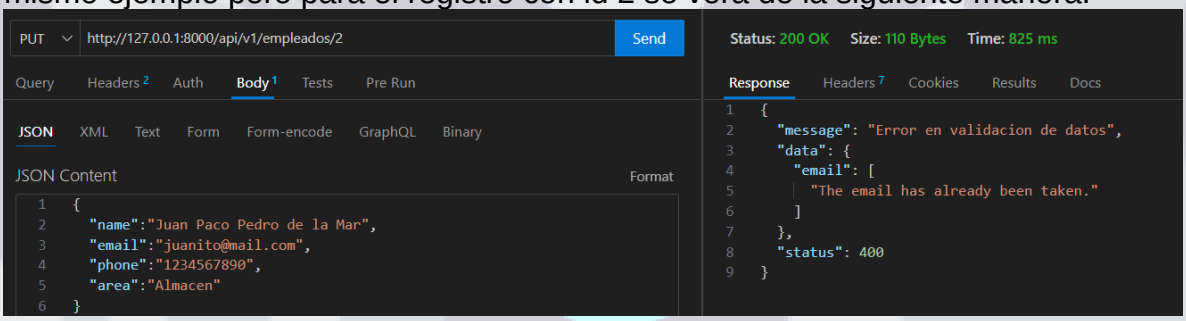
```
use App\Http\Controllers\Api\employedController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;

Route::prefix('v1')->group(function () {
    Route::get('/empleados', [employedController::class,'index']);
    Route::post('/empleados', [employedController::class,'store']);
    Route::get('/empleados/{id}', [employedController::class,'show']);
    Route::delete('/empleados/{id}', [employedController::class,'destroy']);
    Route::put('/empleados/{id}', [employedController::class,'update']);
});
```

Para las pruebas intentaremos realizar la actualización a un registro que eliminamos con anterioridad.
NOTA: PUT vs PATCH; El método PUT se debe utilizar cuando sea necesario hacer un reemplazo total de un recurso. Es decir, si tenemos una tabla Usuarios y queremos actualizar todos los datos de un usuario lo deberíamos implementar.

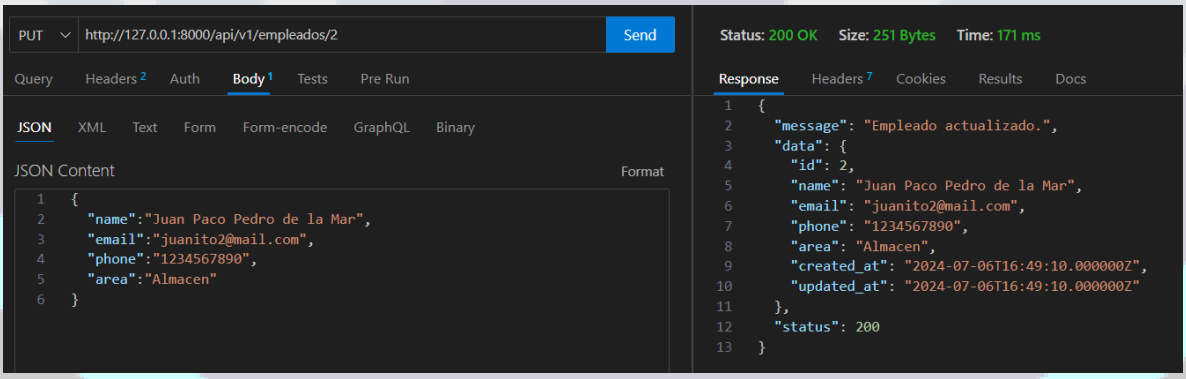
Por su parte el método PATCH debe ser utilizado cuando se quiere modificar un solo campo. En este caso, si solo queremos actualizar el Email de uno de nuestros usuarios, PATCH seria el recomendado.
Al realizar la ejecución del ejemplo, se muestra de la siguiente manera:



En el caso anterior estamos intentando realziar la actualización de datos al registro con el id 1, pero dicho registro ya había sido eliminado. Cuando se realiza el mismo ejemplo pero para el registro con id 2 se vera de la siguiente manera:



En este caso nos arroja un error el cual nos hace ver que este correo ya fue ocupado por alguien, para que fuera valida la actualización entonces se deberá de colocar uno diferente.



4.3.4.2 Ejemplo con PATCH

Como se mencionó el uso de PUT y PATCH varia de la necesidad, por ejemplo para PUT es necesario colocar todos los datos actuales más los campos que se quieren actualizar, no así con PATCH.
Para este ejemplo en específico se creará un method 'updateAttribute' dentro del controller para generar la lógica de la actualización parcial de datos.

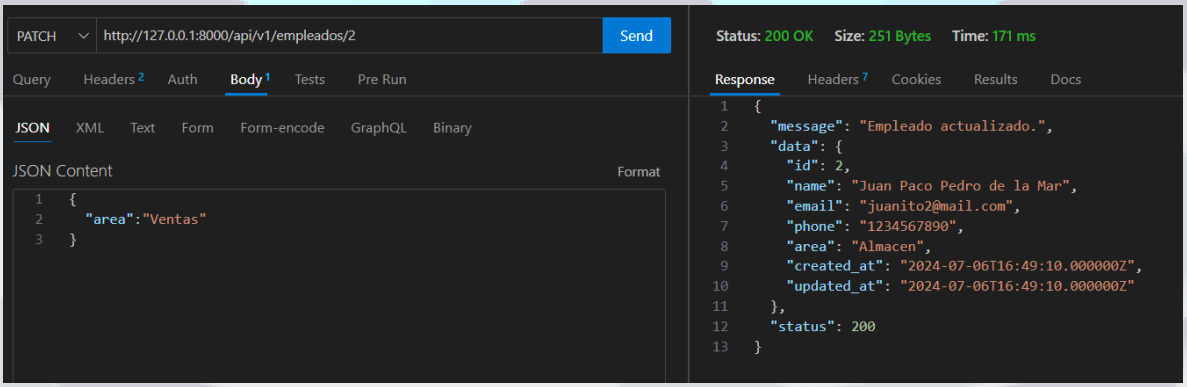
```php
namespace App\Http\Controllers\Api;
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Models\Employed;
use Illuminate\Support\Facades\Validator;

class employedController extends Controller
{
    public function index()
    {…}
    public function store(Request $request)
    {…}
    public function show(string $id)
    {…}
    public function destroy(string $id)
    {…}
    public function update(Request $request, string $id)
    {…}
    public function updateAttribute (Request $request, string $id){
        $employed = Employed::find($id);
        if(!$employed){
            $data=[
                'message'=>'Error empleado no encontrado',
                'data'=>array(),=>404
            ];
        }else{
            $validator = Validator::make($request->all(),[
                'name'=>'max:255',
                'email'=>'email|unique:employed',
                'phone'=>'digits:10',
                'area'=>'in:Almacen,Ventas,Administrativo',
            ]);
            $data=[
                'message'=>'Error al ejecutar creación de empleado',
                'data'=>$validator->errors(),'status'=>400
            ];
```

```
            if($validator->fails()){
                $data=[
                    'message'=>'Error en validacion de datos',
                    'data'=>$validator->errors(),'status'=>400
                ];
            }else{
                if($request->has('name')){ $employed->name=$request->name; }
                if($request->has('email')){ $employed->email=$request->email; }
                if($request->has('phone')){ $employed->phone=$request->phone; }
                if($request->has('area')){ $employed->area=$request->area; }
                $employed->save();
                $data=[
                    'message'=>'Empleado actualizado.',
                    'data'=>$employed, 'status'=>200
                ];
            }
        }
        return response()->json($data);
    }
}
```

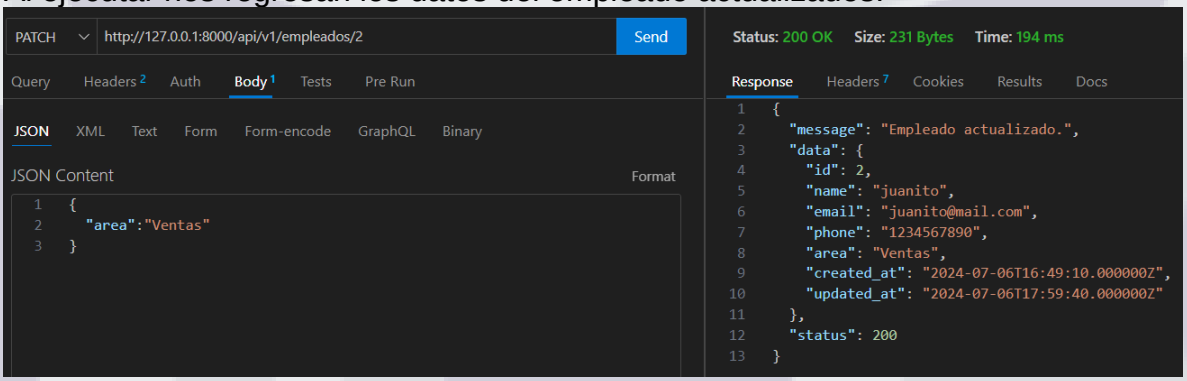
Y dentro de las rutas se debe de especificar el mismo:

```
use App\Http\Controllers\Api\employedController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;
Route::prefix('v1')->group(function () {
    Route::get('/empleados', [employedController::class,'index']);
    Route::post('/empleados', [employedController::class,'store']);
    Route::get('/empleados/{id}', [employedController::class,'show']);
    Route::delete('/empleados/{id}', [employedController::class,'destroy']);
    Route::put('/empleados/{id}', [employedController::class,'update']);
    Route::patch('/empleados/{id}', [employedController::class,'updateAttribute']);
});
```

Al realizar las pruebas se puede notar que el método es ahora ‘PATCH’ y que en el JSON de envió solo se coloca el área:



Al ejecutar nos regresan los datos del empleado actualizados:



5 Pruebas

Laravel proporciona un diseño de directorio predeterminado estructurado. El directorio raíz de tu proyecto Laravel contiene un directorio tests con los subdirectorios Feature y Unit . Este diseño facilita la separación de los distintos tipos de pruebas y mantiene un entorno de pruebas limpio y organizado.

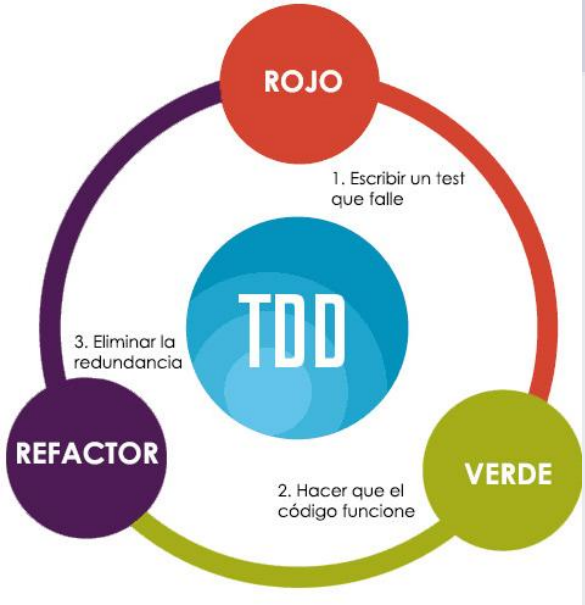
Puedes realizar varios tipos de pruebas en tu aplicación Laravel:

- Pruebas unitarias: se centran en componentes individuales de tu código, como clases, métodos y funciones. Estas pruebas permanecen aisladas de la aplicación Laravel y verifican que determinadas unidades de código funcionan como se espera. Ten en cuenta que las pruebas definidas en el directorio tests/Unit no inician la aplicación Laravel, lo que significa que no pueden acceder a la base de datos ni a otros servicios que ofrece el framework.
- Pruebas de características: validan la funcionalidad más amplia de tu aplicación. Estas pruebas simulan peticiones y respuestas HTTP,

permitiéndote probar rutas, controladores y la integración de varios componentes. Las pruebas de características ayudan a garantizar que las distintas partes de tu aplicación funcionan juntas como se espera.

- Pruebas de navegador: van más allá automatizando las interacciones del navegador. Las pruebas utilizan Laravel Dusk, una herramienta de pruebas y automatización del navegador, para simular las interacciones del usuario, como rellenar formularios y hacer clic en botones. Las pruebas de navegador son cruciales para validar el comportamiento de tu aplicación y la experiencia del usuario en navegadores del mundo real.

El desarrollo dirigido por pruebas (TDD, Test-driven development) es un enfoque de desarrollo de software que hace hincapié en la realización de pruebas antes de implementar el código. Este enfoque sigue un proceso conocido como ciclo rojo-verde-refactorización.



- Fase roja: Escribe una nueva prueba para definir la funcionalidad o una mejora de una existente antes de implementar el código real. La prueba debe fallar (como significa «rojo») porque no hay código correspondiente para hacerla pasar.
- Fase verde: Escribe el código suficiente para que la prueba que falla pase, convirtiéndola de roja a verde. El código no será óptimo, pero cumple los requisitos del caso de prueba correspondiente.
- Fase de refactorización: Refactoriza el código para mejorar su legibilidad, mantenimiento y rendimiento sin cambiar su comportamiento. En esta fase, puedes realizar cómodamente cambios en el código sin preocuparte por los

problemas de regresión, ya que los casos de prueba existentes los detectan.

5.1 Pruebas de funcionalidad

5.1.2 Configurar entorno de pruebas

Laravel viene con PHPUnit preconfigurado para realizar pruebas unitarias. Primero, asegúrate de que tienes PHPUnit instalado. Puedes instalarlo mediante Composer si aún no lo tienes:

```
composer require --dev phpunit/phpunit
```

Es importante verificar las variables gobales de pruebas ubicadas en: .\phpunit.xml ya que en esta se deberá de definir la configuración de base de datos que funionara para hacer pruebas los valores por defecto son:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="vendor/phpunit/phpunit/phpunit.xsd"
         bootstrap="vendor/autoload.php"
         colors="true"
>
    <testsuites>
        <testsuite name="Unit">
            <directory>tests/Unit</directory>
        </testsuite>
        <testsuite name="Feature">
            <directory>tests/Feature</directory>
        </testsuite>
    </testsuites>
    <source>
        <include>
            <directory>app</directory>
        </include>
    </source>
    <php>
        <env name="APP_ENV" value="testing"/>
        <env name="APP_MAINTENANCE_DRIVER" value="file"/>
        <env name="BCRYPT_ROUNDS" value="4"/>
        <env name="CACHE_STORE" value="array"/>
        <!-- <env name="DB_CONNECTION" value="sqlite"/> -->
        <!-- <env name="DB_DATABASE" value=":memory:"/> -->
        <env name="MAIL_MAILER" value="array"/>
```

```
        <env name="PULSE_ENABLED" value="false"/>
        <env name="QUEUE_CONNECTION" value="sync"/>
        <env name="SESSION_DRIVER" value="array"/>
        <env name="TELESCOPE_ENABLED" value="false"/>
    </php>
</phpunit>
```

En donde las líneas comentadas deberán de tener el tipo de la base de datos en DB_CONNECTION y el nombre de la base de datos de prueba en DB_DATABASE de lo contrario las pruebas se plancharan en la base de datos original.

5.1.3 Creación de un test

Para crear un test, puedes usar el comando make:test de Artisan. Por ejemplo, para crear un test de ejemplo para un controlador llamado UserController, puedes ejecutar:

```
php artisan make:test UserControllerTest
```

Esto creará un archivo de test en el directorio tests/Feature llamado UserControllerTest.php. El archivo que crea se encuentra en la ruta tests\Feature\UserControllerTest.php y su contenido es el siguiente:

```php
namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

class UserControllerTest extends TestCase
{
    /**
     * A basic feature test example.
     */
    public function test_example(): void
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

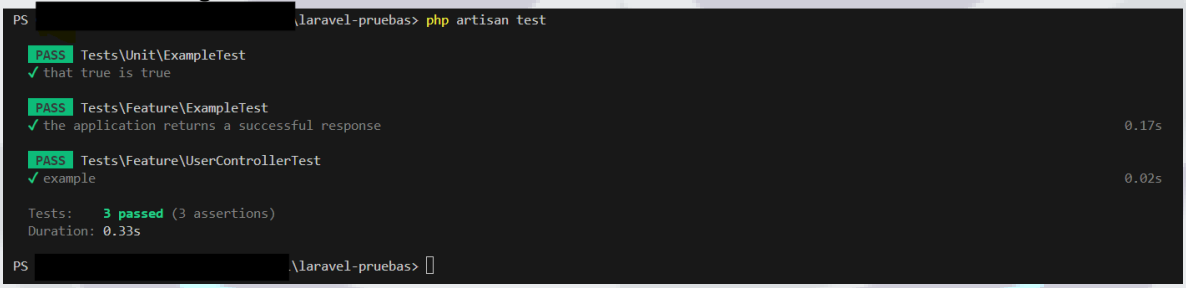
En este ejemplo, la prueba verifica que la ruta raíz (/) devuelve un estado HTTP 200.

5.1.4 Ejecucion de pruebas

Para ejecutar las pruebas, utiliza el siguiente comando en tu terminal:

```
php artisan test
```

Se ejecutaran todas las pruebas que generemos, además, la prueba en si mostrara lo siguiente en terminal



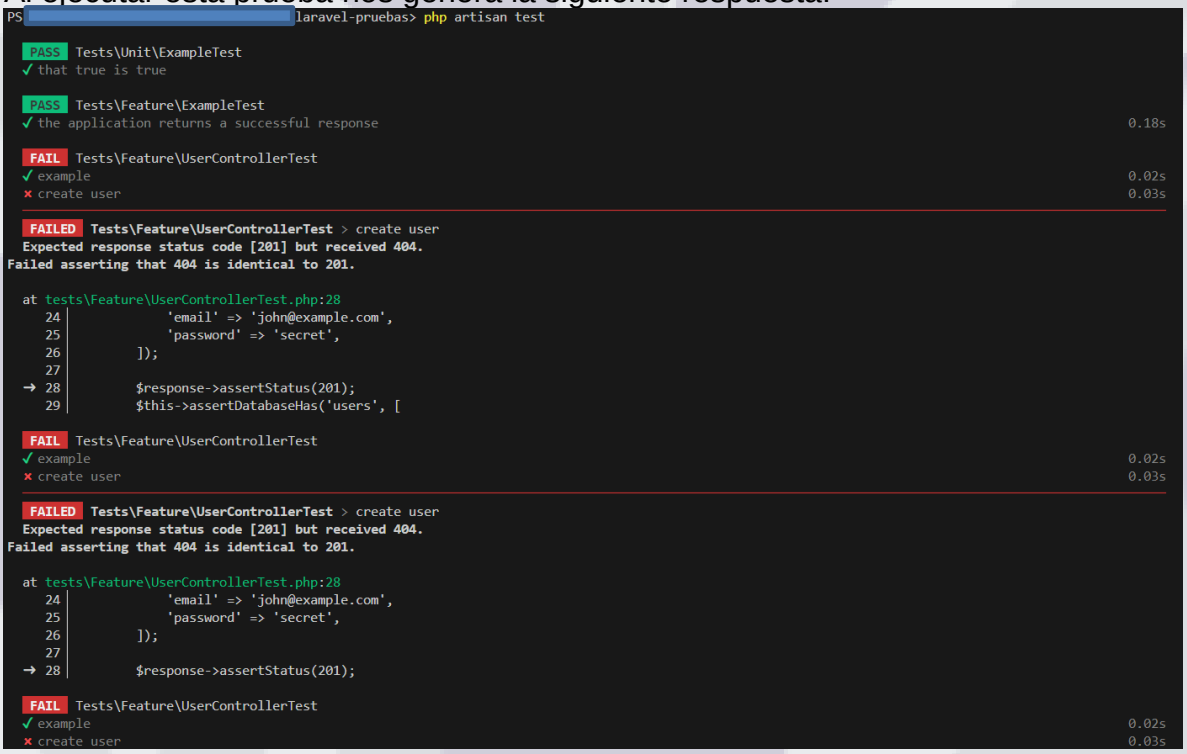
5.1.5 Más sobre pruebas

Puedes añadir más métodos de prueba en tu archivo de prueba para cubrir diferentes aspectos de tu aplicación. Por ejemplo, para probar la creación de un usuario se agrega al archivo de test el method 'test_create_user':

```php
namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

class UserControllerTest extends TestCase
{
    /**
     * A basic feature test example.
     */
    public function test_example(): void
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
    public function test_create_user()
    {
        $response = $this->post('/users', [
            'name' => 'John Doe',
            'email' => 'john@example.com',
            'password' => 'secret',
        ]);

        $response->assertStatus(201);
```

```
        $this->assertDatabaseHas('users', [
            'email' => 'john@example.com',
        ]);
    }
}
```

Al ejecutar esta prueba nos genera la siguiente respuesta:



```
  FAILED  Tests\Feature\UserControllerTest > create user
  Expected response status code [201] but received 404.
Failed asserting that 404 is identical to 201.

  at tests\Feature\UserControllerTest.php:28
     24 |             'email' => 'john@example.com',
     25 |             'password' => 'secret',
     26 |         ]);

  FAIL  Tests\Feature\UserControllerTest
  ✓ example                                                              0.02s
  × create user                                                          0.03s

  FAILED  Tests\Feature\UserControllerTest > create user
  Expected response status code [201] but received 404.
Failed asserting that 404 is identical to 201.

  at tests\Feature\UserControllerTest.php:28
     24 |             'email' => 'john@example.com',

  FAIL  Tests\Feature\UserControllerTest
  ✓ example                                                              0.02s
  × create user                                                          0.03s

  FAILED  Tests\Feature\UserControllerTest > create user
  Expected response status code [201] but received 404.
Failed asserting that 404 is identical to 201.

  at tests\Feature\UserControllerTest.php:28
     24 |             'email' => 'john@example.com',
     25 |             'password' => 'secret',

  FAIL  Tests\Feature\UserControllerTest
  ✓ example                                                              0.02s
  × create user                                                          0.03s

  FAILED  Tests\Feature\UserControllerTest > create user
  Expected response status code [201] but received 404.
Failed asserting that 404 is identical to 201.

  at tests\Feature\UserControllerTest.php:28
  ✓ example                                                              0.02s
  × create user                                                          0.03s

  FAILED  Tests\Feature\UserControllerTest > create user
  Expected response status code [201] but received 404.
Failed asserting that 404 is identical to 201.

  FAILED  Tests\Feature\UserControllerTest > create user
  Expected response status code [201] but received 404.
Failed asserting that 404 is identical to 201.

  at tests\Feature\UserControllerTest.php:28
     24 |             'email' => 'john@example.com',
     25 |             'password' => 'secret',
     26 |         ]);
     27 |
  ➜  28 |         $response->assertStatus(201);
     29 |         $this->assertDatabaseHas('users', [
     30 |             'email' => 'john@example.com',
     31 |         ]);
     32 |     }


  Tests:    1 failed, 3 passed (4 assertions)
  Duration: 0.36s
```

Durante esta prueba se generaron las secciónes rojas que nos dan a entender que es lo que nos hace falta.

Para este ejemplo se realizó el controller y se generaron las rutas siguientes:

- Controller (\app\Http\Controllers\UserController.php)

```php
namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;
```

```php
class UserController extends Controller
{
    /**
     * Display a listing of the resource.
     */
    public function index()
    {
        $users = User::all();
        return view('users.index', compact('users'));
    }

    /**
     * Show the form for creating a new resource.
     */
    public function create()
    {
        return view('users.create');
    }

    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(),[
            'name'=>'required|max:255',
            'email'=>'required|email|unique:users',
            'password'=>'required|digits:10'
        ]);
        $data=[
            'message'=>'Error al ejecutar creación de empleado',
            'data'=>$validator->errors(),
            'status'=>400
        ];
        if($validator->fails()){
            $data=[
                'message'=>'Error en validacion de datos',
                'data'=>$validator->errors(),
                'status'=>400
            ];
        }else{
            $user = User::create([
```

```
                'name'=>$request->name,
                'email'=>$request->email,
                'password'=>$request->password
            ]);
            if(!$user){
                $data=[
                    'message'=>'Error al crear empleado',
                    'status'=>500
                ];
            }else{
                $data=[
                    'message'=>'Estudiante creado.',
                    'data'=>$user,
                    'status'=>200
                ];
            }
        }
        return response()->json($data);
    }

    /**
     * Display the specified resource.
     */
    public function show(string $id)
    {
        $user = User::find($id);
        return view('users.show',compact('post'));
    }

    /**
     * Show the form for editing the specified resource.
     */
    public function edit(string $id)
    {
        $user = User::find($id);
        return view('users.edit',compact('post'));
    }

    /**
     * Update the specified resource in storage.
     */
    public function update(Request $request, User $user)
```

```
    {
        $validatedData = $request->validate([
            'name' => 'required|max:255',
            'email' => 'required',
            'password' => 'required',
        ]);

        $user->update($validatedData);

        return redirect()->route('users.index')->with('success', 'Post
actualizado con éxito');
    }

    /**
     * Remove the specified resource from storage.
     */
    public function destroy(string $id)
    {
        $user = User::find($id);
        $user->delete();
        return redirect()->route('users.index')->with('success','Post
eliminado exitosamente.');
    }
}
```

- Rutas (\routes\web.php)

```
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\UserController;

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

Al ejecutar nuevamente la prueba unitaria nos arroja lo siguiente:

```
PS                                  \laravel-pruebas> php artisan test

  PASS  Tests\Unit\ExampleTest
  ✓ that true is true

  PASS  Tests\Feature\ExampleTest
  ✓ the application returns a successful response                    0.18s

  PASS  Tests\Feature\UserControllerTest
  ✓ create user                                                      0.41s

  Tests:    3 passed (4 assertions)
  Duration: 0.73s
```

Con esto se puede verificar que la prueba paso correctamente.

6 Proyecto final

6.1 Problematica

Una empresa requiere de un sistema que permita asignar tareas a sus empleados, cada empleado puede ver sus propias tareas pero no puede ver la de los demás, solo el usuario root y el administrador podrán ver todas las tareas, asignar tareas y actualizarlas, pero solo el usuario root podrá eliminar tareas.
Este sistema compartirá las tareas de cada empleado a otro sistema por API por lo que deberá de tener solo disponible una API que al darle el id de un empleado regrese los datos del empleado y sus tareas.
El cliente requiere que se realicen las pruebas necesarias y se obtengan comprobables de las mismas.

6.2 Desarrollo

Se crea el proyecto de laravel

```
composer create-project --prefer-dist laravel/laravel proyecto-final
```

Se crea una base de datos para dicho proyecto y una base de datos para las pruebas.
'Productivo'

```
CREATE DATABASE proyecto_final CHARACTER SET utf8mb4 COLLATE
utf8mb4_unicode_ci;
```

'Pruebas'

```
CREATE DATABASE proyecto_final_test CHARACTER SET utf8mb4 COLLATE
utf8mb4_unicode_ci;
```

Se deberán de configurar los archivos *proyecto-final\.env* y *proyecto-final\phpunit.xml*
Para el archivo '.env':

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

Para el archivo 'phpunit.xml'

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="vendor/phpunit/phpunit/phpunit.xsd"
         bootstrap="vendor/autoload.php"
         colors="true">
    <testsuites>
        <testsuite name="Unit">
            <directory>tests/Unit</directory>
```

```
        </testsuite>
        <testsuite name="Feature">
            <directory>tests/Feature</directory>
        </testsuite>
    </testsuites>
    <source>
        <include>
            <directory>app</directory>
        </include>
    </source>
    <php>
        <env name="APP_ENV" value="testing"/>
        <env name="APP_MAINTENANCE_DRIVER" value="file"/>
        <env name="BCRYPT_ROUNDS" value="4"/>
        <env name="CACHE_STORE" value="array"/>
        <env name="DB_CONNECTION" value="mysql"/>
        <env name="DB_DATABASE" value="proyecto_final_test"/>
        <env name="MAIL_MAILER" value="array"/>
        <env name="PULSE_ENABLED" value="false"/>
        <env name="QUEUE_CONNECTION" value="sync"/>
        <env name="SESSION_DRIVER" value="array"/>
        <env name="TELESCOPE_ENABLED" value="false"/>
    </php>
</phpunit>
```

Se deberán de migrar las tablas bases creadas con el proyecto.

```
php artisan migrate
```

Posteriormente se debe de verificar que este instalado breeze, sanctum y spatie para permisos, roles y autenticación.

```
composer require laravel/breeze --dev
composer require laravel/sanctum
php artisan breeze:install
composer require spatie/laravel-permission
```

Para la configuración de ‘breeze’ se seleccióno lo siguiente:

```
PS C:\...\proyecto-final> php artisan breeze:install

  Which Breeze stack would you like to install?
  Blade with Alpine ..... blade
  Livewire (Volt Class API) with Alpine ..... livewire
  Livewire (Volt Functional API) with Alpine ..... livewire-functional
  React with Inertia ..... react
  Vue with Inertia ..... vue
  API only ..... api
> blade

  Would you like dark mode support? (yes/no) [no]
> no

  Which testing framework do you prefer? [PHPUnit]
  Pest ..... 0
  PHPUnit ..... 1
> 1

  INFO  Installing and building Node dependencies.
```

Posteriormente se deberá de publicar la aplicación y actualziar las migraciónes necesarias.

```
php artisan vendor:publish --provider="Spatie\Permission\PermissionServiceProvider"
php artisan migrate
```

Como base Laravel nos genera una tabla de usuarios con los siguientes campos:

id	name	email	email_verified_at	password	remember_token	created_at	updated_at

Para este ejercicio agregaremos apellidos y teléfono como atributos, para ello crearemos una primera migración de la siguiente manera:

```
php artisan make:migration add_lastname_column_on_users_table --table=users
```

Este comando nos generara una migración a la cual se deberá de colocar lo necesario para ser específica con la información que modificara, de tal manera que se vera como a continuación:

```php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->string('lastname')->after('name');
        });
    }
    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::table('users', function (Blueprint $table) {
```

```
            $table->dropColumn('lastname');
        });
    }
};
```

Se realizan los mismos pasos para agregar el campo teléfono.

```
php artisan make:migration add_phone_column_on_users_table --table=users
```

Se define la migración

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->string('phone',10)->after('password');
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->dropColumn('phone');
        });
    }
};
```

Una vez definidos los datos de cada migración se deberá de ejecutar dicha migración.

```
php artisan migrate
```

```
PS                          proyecto-final> php artisan migrate

  INFO  Running migrations.

  2024_07_06_214331_add_lastname_column_on_users_table ..... 11.03ms DONE
  2024_07_06_214728_add_phone_column_on_users_table ..... 6.85ms DONE
```

Con esto podemos verificar que efectivamente se modifico la estructura de la tabla.

Field	Type	Null	Key	Default	Extra
id	bigint unsigned	NO	PRI	NULL	auto_increment
name	varchar(255)	NO		NULL	
lastname	varchar(255)	NO		NULL	
email	varchar(255)	NO	UNI	NULL	
email_verified_at	timestamp	YES		NULL	
password	varchar(255)	NO		NULL	
phone	varchar(10)	YES		NULL	
remember_token	varchar(100)	YES		NULL	
created_at	timestamp	YES		NULL	
updated_at	timestamp	YES		NULL	

De la misma manera se deberán de configurar estos cambios en el modelo dentro del array de datos 'fillable'.

```php
namespace App\Models;

// use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Spatie\Permission\Traits\HasRoles;

class User extends Authenticatable
{
    use HasFactory, Notifiable, HasRoles;

    /**
     * The attributes that are mass assignable.
     *
     * @var array<int, string>
     */
    protected $fillable = [
        'name',
        'lastname',
        'email',
        'password',
        'phone',
    ];

    /**
     * The attributes that should be hidden for serialization.
     *
     * @var array<int, string>
```

```
     */
    protected $hidden = [
        'password',
        'remember_token',
    ];

    /**
     * Get the attributes that should be cast.
     *
     * @return array<string, string>
     */
    protected function casts(): array
    {
        return [
            'email_verified_at' => 'datetime',
            'password' => 'hashed',
        ];
    }
}
```

Para poder tner un mejor manejo de usuarios primero crearemos el controller de user que nos permitirá realizar las operaciones del CRUD.

```
php artisan make:controller UserController --resource
```

Una vez creado el controller se definen toda la lógica de la misma:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Models\User;
use Illuminate\Support\Facades\Validator;

class UserController extends Controller
{
    /**
     * Display a listing of the resource.
     */
    public function index()
    {
        $users = User::all();
        return view('users.index', compact('users'));
    }
```

```php
    /**
     * Show the form for creating a new resource.
     */
    public function create()
    {
        return view('users.create');
    }

    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(),[
            'name'=>'required|max:255',
            'lastname'=>'required|max:255',
            'email'=>'required|email|unique:users',
            'password'=>'required|digits:10',
            'phone'=>'required|max:10',
        ]);
        $data=[
            'message'=>'Error al ejecutar creación de empleado',
            'data'=>$validator->errors(),
            'status'=>400
        ];
        if($validator->fails()){
            $data=[
                'message'=>'Error en validacion de datos',
                'data'=>$validator->errors(),
                'status'=>400
            ];
        }else{
            $user = User::create([
                'name'=>$request->name,
                'lastname'=>$request->lastname,
                'email'=>$request->email,
                'password'=>$request->password,
                'phone'=>$request->phone
            ]);
            if(!$user){
                $data=[
                    'message'=>'Error al crear empleado',
```

```
                'status'=>500
            ];
        }else{
            $data=[
                'message'=>'Estudiante creado.',
                'data'=>$user,
                'status'=>200
            ];
        }
    }
    return response()->json($data);
}

/**
 * Display the specified resource.
 */
public function show(string $id)
{
    $user = User::find($id);
    return view('users.show',compact('user'));
}

/**
 * Show the form for editing the specified resource.
 */
public function edit(string $id)
{
    $user = User::find($id);
    return view('users.edit',compact('user'));
}

/**
 * Update the specified resource in storage.
 */
public function update(Request $request, User $user)
{
    $validatedData = $request->validate([
        'name' => 'required|max:255',
        'lastname'=>'required|max:255',
        'email'=>'required|email|unique:users',
        'password'=>'required|digits:10',
        'phone'=>'required|max:10',
```

```
        ]);

        $user->update($validatedData);

        return redirect()->route('users.index')->with('success', 'Usuario
actualizado con éxito');
    }
    public function updateAttribute (Request $request, string $id){
        $user = User::find($id);
        if(!$user){
            $data=[
                'message'=>'Error empleado no encontrado',
                'data'=>array(),
                'status'=>404
            ];
        }else{
            $validator = Validator::make($request->all(),[
                'name'=>'max:255',
                'lastname'=>'max:255',
                'email'=>'email|unique:user',
                'password'=>'required|digits:10',
                'phone'=>'digits:10',
            ]);
            $data=[
                'message'=>'Error al ejecutar creación de empleado',
                'data'=>$validator->errors(),
                'status'=>400
            ];
            if($validator->fails()){
                $data=[
                    'message'=>'Error en validacion de datos',
                    'data'=>$validator->errors(),
                    'status'=>400
                ];
            }else{
                if($request->has('name')){ $user->name=$request->name; }
                if($request->has('lastname')){ $user->lastname=$request-
>lastname; }
                if($request->has('email')){ $user->email=$request->email; }
                if($request->has('password')){ $user->password=$request-
>password; }
                if($request->has('phone')){ $user->phone=$request->phone; }
```

```
                $user->save();
                $data=[
                    'message'=>'Empleado actualizado.',
                    'data'=>$user,
                    'status'=>200
                ];
            }
        }
        return response()->json($data);
    }
    /**
     * Remove the specified resource from storage.
     */
    public function destroy(string $id)
    {
        $user = User::find($id);
        $user->delete();
        return redirect()->route('users.index')->with('success','Usuario
eliminado exitosamente.');
    }
}
```

Aprovechando la creación se sugiere instalar phpunit en caso de no tenerlo instalado para futuramente realizar pruebas.

```
composer require --dev phpunit/phpunit
```

Posteriormentre se creara un seeder para registrar a nuestro usuario root y otro para registrar los primeros roles y permisos.

```
php artisan make:seeder UserSeeder
php artisan make:seeder RolesSeeder
php artisan make:seeder PermissionsSeeder
```

Dentro del seeder de usuario se agrega lo siguiente:

```
namespace Database\Seeders;

use App\Models\User;
use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

class UserSeeder extends Seeder
```

```
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        $user = User::create([
            'name' => 'Miguel',
            'lastname' => 'Castro',
            'email' => ejemplo@mail.com',
            'password' => bcrypt('toor'),
            'phone' => '123456789',
        ]);
    }
}
```

En el seeder de permisos se coloca lo siguiente:

```
namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Spatie\Permission\Models\Permission;
use Spatie\Permission\Models\Role;

class PermissionsSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        Permission::create(['name' => 'read_user']);
        Permission::create(['name' => 'create_user']);
        Permission::create(['name' => 'update_user']);
        Permission::create(['name' => 'delete_user']);

        Permission::create(['name' => 'read_permission']);
        Permission::create(['name' => 'create_permission']);
        Permission::create(['name' => 'update_permission']);
        Permission::create(['name' => 'delete_permission']);

        Permission::create(['name' => 'read_rol']);
```

```
        Permission::create(['name' => 'create_rol']);
        Permission::create(['name' => 'update_rol']);
        Permission::create(['name' => 'delete_rol']);

        Permission::create(['name' => 'read_task']);
        Permission::create(['name' => 'create_task']);
        Permission::create(['name' => 'update_task']);
        Permission::create(['name' => 'delete_task']);

        Permission::create(['name' => 'read_my_task']);
        Permission::create(['name' => 'create_my_task']);
        Permission::create(['name' => 'update_my_task']);
        Permission::create(['name' => 'delete_my_task']);
    }
}
```

En el seeder de rol se coloca lo siguiente:

```
namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Spatie\Permission\Models\Role;

class RolesSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        Role::create(['name' => 'Root']);
        Role::create(['name' => 'Admin']);
        Role::create(['name' => 'Empleados']);
    }
}
```

Posterior mente se ejecutan estos seeders en el orden siguiente:

```
php artisan db:seed --class=RolesSeeder
php artisan db:seed --class=PermissionsSeeder
php artisan db:seed --class=UserSeeder
```

En la terminal deberá de irnos arrojando el mensaje siguiente:

```
PS [redacted].\proyecto-final> php artisan db:seed --class=RolesSeeder

  INFO  Seeding database.

PS [redacted]\proyecto-final> php artisan db:seed --class=PermissionsSeeder

  INFO  Seeding database.

PS [redacted]proyecto-final> php artisan db:seed --class=UserSeeder

  INFO  Seeding database.
```

En base de datos se ve asi:

```
SELECT * FROM proyecto_final.roles;
```

id	name	guard_name	created_at	updated_at
1	Root	web	2024-07-07 17:14:21	2024-07-07 17:14:21
2	Admin	web	2024-07-07 17:14:21	2024-07-07 17:14:21
3	Empleados	web	2024-07-07 17:14:21	2024-07-07 17:14:21

```
SELECT * FROM proyecto_final.permissions;
```

id	name	guard_name	created_at	updated_at
1	read_user	web	2024-07-07 17:16:03	2024-07-07 17:16:03
2	create_user	web	2024-07-07 17:16:03	2024-07-07 17:16:03
3	update_user	web	2024-07-07 17:16:03	2024-07-07 17:16:03
4	delete_user	web	2024-07-07 17:16:03	2024-07-07 17:16:03
5	read_permission	web	2024-07-07 17:16:03	2024-07-07 17:16:03
6	create_permission	web	2024-07-07 17:16:03	2024-07-07 17:16:03
7	update_permission	web	2024-07-07 17:16:03	2024-07-07 17:16:03
8	delete_permission	web	2024-07-07 17:16:03	2024-07-07 17:16:03
9	read_rol	web	2024-07-07 17:16:03	2024-07-07 17:16:03
10	create_rol	web	2024-07-07 17:16:04	2024-07-07 17:16:04
11	update_rol	web	2024-07-07 17:16:04	2024-07-07 17:16:04
12	delete_rol	web	2024-07-07 17:16:04	2024-07-07 17:16:04
13	read_task	web	2024-07-07 17:16:04	2024-07-07 17:16:04
14	create_task	web	2024-07-07 17:16:04	2024-07-07 17:16:04
15	update_task	web	2024-07-07 17:16:04	2024-07-07 17:16:04
16	delete_task	web	2024-07-07 17:16:04	2024-07-07 17:16:04
17	read_my_task	web	2024-07-07 17:16:04	2024-07-07 17:16:04
18	create_my_task	web	2024-07-07 17:16:04	2024-07-07 17:16:04
19	update_my_task	web	2024-07-07 17:16:04	2024-07-07 17:16:04
20	delete_my_task	web	2024-07-07 17:16:04	2024-07-07 17:16:04

```
SELECT * FROM proyecto_final.users;
```

id	name	lastname	email	email_verified_at	password	phone	remember_token	created_at	updated_at
1	Miguel	Castro	[redacted]com	NULL	$2y$12$XrFkt0FjMzK9PnhQJpShROq0UstrNLSlZ...	123456789	NULL	2024-07-07 17:17:53	2024-07-07 17:17:53

Ahora falta crear un seeder que nos permita asignarle a un rol los permisos que se requieren y a su vez asinar el rol al usuario creado, para este ejemplo será un seeder que agrege lo necesario para el usuario root.

```
php artisan make:seeder UserRolPermissionSeeder
```

```
PS [redacted]\proyecto-final> php artisan make:seeder UserRolPermissionSeeder

  INFO  Seeder [C:\laragon\www\cursos\CursoLaravel\proyecto-final\database\seeders\UserRolPermissionSeeder.php] created successfully.
```

Y el contenido de dicho seeder es el siguiente:

```php
namespace Database\Seeders;

use Illuminate\Database\Seeder;
```

```php
use Spatie\Permission\Models\Permission;
use Spatie\Permission\Models\Role;
use App\Models\User;

class UserRolPermissionSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        $root = Role::find(1);
        $root->givePermissionTo(Permission::all());
        User::find(1)->assignRole($root);
    }
}
```

Y se ejecuta el seeder.

```
php artisan db:seed --class=UserRolPermissionSeeder
```

```
PS                                    \proyecto-final> php artisan db:seed --class=UserRolPermissionSeeder

  INFO  Seeding database.
```

Ahora se deberán de preparar las visas de CRUD para cada acción.