API REST Con Laravel 9 y 10

Contenidos

1 API REST con Laravel

Vamos a ver las posibilidades de escribir una `API REST` con Laravel 10. Podemos hacer un `CRUD` como hemos hecho antes con el blog. Nos daremos cuenta de que, al final, nos va a quedar algo mucho más sencillo, puesto que no tendremos vistas ni plantillas, ni componentes... Sólo **rutas**, y **puntos de acceso** —=endpoints=—.

1.1 API en pocas palabras

Una **Interfaz de Programación de Aplicaciones** permite a dos sistemas comunicarse uno con otro. Proporciona el lenguaje y el contrato de como interactuan los dos sistemas. Cada `API` tiene documentación y especificaciones que determinan cómo se puede transferir la información.

Las `API` (WEB) generalmente se clasifican como `SOAP` o `REST` y ambas se usan para acceder a los servicios web.

`SOAP` se basa únicamente en `XML` para proporcionar servicios de mensajería, mientras que `REST` ofrece un método más ligero, utilizando direcciones `URL` en la mayoría de los casos para recibir o enviar información.

Vamos a crear `APIs REST`.

Para aprovechar al máximo las `API`, las empresas las usan para:

- Integrarse con `API` de terceros.
- Para uso interno.
- Para exponerla para uso externo.

1.1.1 ¿Que es un EndPoint?

Brevemente un endPoint es un extremo de un canal de comunicación. Cuando una API interactúa con otro sistema, los puntos de contacto de esta comunicación se consideran endpoints.

Cuando una API solicita información de una aplicación web o un servidor web, recibirá una respuesta. El lugar donde las `API` envían solicitudes y donde vive el recurso se denomina `endpoint`.

Para las `API`, un `endpoint` puede incluir una `URL` de un *servidor*. Cada `endpoint` es la ubicación desde la cual las API pueden acceder a los recursos que necesitan para llevar a cabo su función.

1.2 Laravel Sail

Vamos a usar de nuevo `Laravel Sail`, que para montar un entorno de desarrollo es de lo más cómodo y fácil.

Laravel Sail es una interfaz de línea de comandos liviana para interactuar con el entorno de desarrollo `Docker` de `Laravel`. Proporciona un excelente punto de partida para crear una aplicación Laravel usando `PHP`, `MySQL` y `Redis` sin requerir experiencia previa en `Docker`.

`Sail` en esencia es un archivo `docker-compose.yml` y el script sail que se almacena en la raíz de tu proyecto.

El archivo `docker-compose.yml` define una variedad de contenedores de `Docker`.

`Sail` es automáticamente instalado con todas las aplicaciones nuevas de `Laravel`.

2 Aplicación Artículos con imágenes (blog con entradas de texto y/o imágenes)

Al crear una nueva aplicación Laravel a través de Sail, puedes elegir qué servicios deben configurarse en el archivo docker-compose.yml de tu nueva aplicación.

Ejemplo:

```
curl -s "https://laravel.build/laravel10-api?with=mysql,redis" | bash
```

En el ejemplo anterior se va a instalar mysql y redis, si no especificas qué servicios quieres por defecto se instala el siguiente stack: mysql, redis, meilisearch, mailhog y selenium.

Por defecto los comando sail son invocados desde vendor/bin/sail:

```
cd laravel10-api && ./vendor/bin/sail up
```

Para evitar escribir repetidamente vendor/bin/sail podemos definir un alias para invocar comandos de sail más fácilmente:

```
alias sail='bash vendor/bin/sail'
```

2.1 Iniciar y detener Sail

Antes de iniciar Sail, debes asegurarte que no hay otros servicios de bases de datos o servidores web corriendo en tu máquina local ya que esto puede arrojar errores al ejecutar Sail.

Para iniciar todos los contenedores Docker definidos en el archivo docker-compose.yml, debes ejecutar el comando up:

```
sail up
```

La primera vez que corras Sail up, los contenedores serán construidos en tu máquina; esto puede tomar varios minutos pero después los subsecuentes inicios Sail comenzará mucho más rápido.

Para iniciar los contenedores en modo background deberías ejecutar sail en modo "detached":

```
sail up -d
```

Una vez que se han iniciado los contenedores Docker de la aplicación puedes acceder con tu navegador web en la dirección: `http://localhost`.

2.2 Rutas, modelos, migraciones y controllers

- Rutas: Las rutas son el punto de entrada de tu aplicación web; sin estas no se podría interactuar con el usuario final.
- Las peticiones HTTP: A veces son llamados HTTP verbs. Son un conjunto de métodos de petición para indicar que se desea realizar para un recurso determinado.
  - GET: solicita un recurso (o lista de recursos).
  - HEAD: pide una respuesta idéntica a la de una petición GET, pero sin el cuerpo de la respuesta.
  - POST: crea un recurso.
  - PUT: modifica un recurso.
  - DELETE: borra un recurso.

  - OPTIONS: describe las opciones de comunicación para el recurso destino.
  - PATCH: es utilizado para aplicar modificaciones parciales a un recurso.

- Definición de rutas: Hay varios archivos donde puede definir las rutas; estos se encuentran en el directorio `routes`. Nosotros vamos a trabajar con el archivo `api.php` dentro del directorio mencionado.

  La forma más simple de definir una ruta es emparejando un «path» con una función anónima `lambda` (o closure, como les gusta a muchos decir hoy en día), como se muestra a continuación:

```php
<?php
Route::get('/', function(){
    return "Hello world";
});
```

  Aquí la ruta `localhost/api/` muestra el mensaje: "Hello world".

  Lo siguiente es continuar con el modelo, la migración y el controlador.

2.3 Modelos

Para este caso vamos a crear un modelo llamado `Post` que guardará información sobre las fotos que queramos con un título y una descripción. Para ello lanzamos el siguiente comando:

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

Al añadir el parámetro `-m`, le estamos indicando que además del modelo queremos que cree el archivo de migración para la base de datos.

Una vez creado el modelo, vamos a la carpeta `database/migrations`, dentro debemos tener un nuevo archivo que como prefijo tendrá la fecha actual (fecha de la creación de la migración) más _create_posts_table.php. Este archivo contendrá la información para crear la tabla posts dentro de nuestra base de datos. Así que para añadir las columnas que necesitamos, lo abrimos y sustituimos el contenido del método up por el siguiente:

```php
<?=
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->foreignId('user_id')->constrained();
        $table->string('title');
        $table->string('image');
        $table->mediumText('description');
        $table->timestamps();
    });
}
```

Como comenté antes, añadimos los campos para guardar el título, la imagen y la descripción. También generamos la relación con la tabla de usuarios para saber de qué usuario es el post. El método `id()` creará el campo para la clave primaria y el método `timestamps()` genera dos campos para guardar la fecha de creación y la fecha de la última actualización.

Ahora debemos abrir el archivo `app/Models/Post.php` y dentro de la clase, añadiremos el siguiente código de tal forma que el archivo nos debería quedar algo así:

```php
<?php
```

```php
namespace App\Models;

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

class Post extends Model
{
    use HasFactory;

    /**
     * Get the user record associated with the post.
     */
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Con el método `user` podremos asociar un usuario con el post y podremos acceder a los datos del usuario cuando recorramos nuestros posts.

Por último, lanzamos el siguiente comando para crear la tabla en la base de datos:

```
sail artisan migrate
```

2.4 Autenticación con token JWT

El siguiente paso es instalar la dependencia de JWT vía composer, para ello solo debemos lanzar el siguiente comando:

```
sail composer require tymon/jwt-auth:*
```

2.4.1 Configuración

Una vez instalado, debemos abrir el archivo `config/app.php`. Ahí veremos que retorna un array enorme. Pues bien, debemos ir a la clave `providers` y añadir la siguiente línea:

```php
<?=
Tymon\JWTAuth\Providers\LaravelServiceProvider::class,
```

De tal forma que quedaría algo así:

```php
<?=
'providers' => [

        /*
         * Laravel Framework Service Providers...
         */
        .
        .
        .
        Illuminate\Validation\ValidationServiceProvider::class,
        Illuminate\View\ViewServiceProvider::class,
        Tymon\JWTAuth\Providers\LaravelServiceProvider::class,
        .
```

```
        .
        .
],
```

Al añadirlo en esta lista, el servicio de `JWT` se cargará automáticamente cada vez que un usuario haga una petición a nuestra API.

Una vez hecho esto, deberemos crear un archivo llamado `config/jwt.php`, para ello, solo debemos lanzar el siguiente comando y este lo creará automáticamente:

```
sail artisan vendor:publish --provider="Tymon\JWTAuth\Providers\LaravelServiceProvider
```

Una vez creado, lanzaremos el siguiente comando para crear una variable de entorno con una `clave` para JWT. Lanzamos el siguiente comando y listo[1]:

```
sail artisan jwt:secret
```

Ahora deberemos modificar la forma en la que nos autenticamos por defecto en Laravel así que lo queremos hacer es abrir el archivo `config/auth.php` y sustituir unas claves y valores por los siguientes:

```
<?=
    .
    .
    .
    'defaults' => [
        'guard' => 'api',
        'passwords' => 'users',
    ],
    .
    .
    .
    'guards' => [
        'api' => [
            'driver' => 'jwt',
            'provider' => 'users',
        ],
    ],
```

En la clave `defaults` lo que hacemos es sustituir `guard` con valor `web` por `api`, ya que el tipo de login que vamos a usar va a ser el de API.

En la clave `guards`, cambiamos la clave `web` por `api`, ya que no la necesitamos y en API le diremos que vamos a usar **el driver de JWT**.

2.4.2 Modelo

Una vez hecho esto, deberemos editar el modelo `User.php` que ya viene por defecto en Laravel, asi que abrimos el archivo `app/Models/User.php` y reemplazamos su contenido por el siguiente:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Tymon\JWTAuth\Contracts\JWTSubject;
use Illuminate\Notifications\Notifiable;
```

```php
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable implements JWTSubject
{
    use Notifiable;
    use HasFactory;

    // Rest omitted for brevity

    /**
     * Get the identifier that will be stored in the subject claim of the JWT.
     *
     * @return mixed
     */
    public function getJWTIdentifier()
    {
        return $this->getKey();
    }

    /**
     * Return a key value array, containing any custom claims to be added to the JWT.
     *
     * @return array
     */
    public function getJWTCustomClaims()
    {
        return [];
    }
}
```

2.4.3 Controlador

Para gestionar las peticiones de login, necesitaremos crear un controlador que se encargue de la autenticación. Para crearlo, lanzamos el siguiente comando:

```
sail  artisan make:controller Api/V1/AuthController
```

De esta forma, crearemos el controlador para gestionar la autenticación en la ruta `app/Http/Controllers/Api/V1` Una vez creado, añadiremos el siguiente código:

```php
<?php

namespace App\Http\Controllers\Api\V1;

use App\Http\Controllers\Controller;

class AuthController extends Controller
{
    /**
     * Create a new AuthController instance.
     *
     * @return void
```

```php
     */
    public function __construct()
    {
        $this->middleware('auth:api', ['except' => ['login']]);
    }

    /**
     * Get a JWT via given credentials.
     *
     * @return \Illuminate\Http\JsonResponse
     */
    public function login()
    {
        $credentials = request(['email', 'password']);

        if (! $token = auth()->attempt($credentials)) {
            return response()->json(['error' => 'Unauthorized'], 401);
        }

        return $this->respondWithToken($token);
    }

    /**
     * Get the authenticated User.
     *
     * @return \Illuminate\Http\JsonResponse
     */
    public function me()
    {
        return response()->json(auth()->user());
    }

    /**
     * Log the user out (Invalidate the token).
     *
     * @return \Illuminate\Http\JsonResponse
     */
    public function logout()
    {
        auth()->logout();

        return response()->json(['message' => 'Successfully logged out']);
    }

    /**
     * Refresh a token.
     *
     * @return \Illuminate\Http\JsonResponse
     */
```

```
    public function refresh()
    {
        return $this->respondWithToken(auth()->refresh());
    }

    /**
     * Get the token array structure.
     *
     * @param  string $token
     *
     * @return \Illuminate\Http\JsonResponse
     */
    protected function respondWithToken($token)
    {
        return response()->json([
            'access_token' => $token,
            'token_type' => 'bearer',
            'expires_in' => auth()->factory()->getTTL() * 60
        ]);
    }
}
```

Este código se encargará de gestionar las distintas rutas que utilizaremos para todo el proceso de autenticación.

Nota:
Para las pruebas podemos cambiar el campo `expires_in`, para no tener que generar un nuevo token cada dos por tres.

2.4.4 Rutas

El siguiente paso que haremos, será configurar las rutas así abrimos el archivo `routes/api.php` y sustituimos su contenido por el siguiente:

```
<?php

use Illuminate\Support\Facades\Route;

Route::group([
    'middleware' => 'api',
    'prefix' => 'v1/auth'

], function ($router) {
    Route::post('login', [\App\Http\Controllers\Api\V1\AuthController::class,
                'login'])->name('login');
    Route::post('logout', [\App\Http\Controllers\Api\V1\AuthController::class,
                 'logout'])->name('logout');
    Route::post('refresh', [\App\Http\Controllers\Api\V1\AuthController::class,
                'refresh'])->name('refresh');
    Route::post('me', [\App\Http\Controllers\Api\V1\AuthController::class,
                'me'])->name('me');
});
```

Como podéis ver, la dependencia de `JWT` en Laravel nos crea varias rutas aunque en nuestro caso sólo utilizaremos la ruta de login para obtener el token de acceso.

También, todas nuestras rutas tendrán un *prefijo* `v1/auth`, de esta forma si luego trabajamos con otra versión, podremos separar como hacer la autenticación de una versión a otra.

2.4.5 Configuración de la BD

Como siempre, deberemos editar el archivo `.env` para añadir las conexiones a la base de datos así que lo abrimos y añadimos la configuración a nuestra `BD`. Nos vale con lo que sail hace, nada que tocar.

Normalmente con Sail no hay que crear la BD, pero a veces se tiene que crear la base de datos con el nombre que queráis en vuestro cliente de MySQL para que todo funcione correctamente.

El siguiente paso es crear unos cuantos usuarios para probar que todo funciona correctamente. Para ello, vamos al archivo `database/seeders/DatabaseSeeder.php` y descomentamos la línea para crear los usuarios. De esta forma podremos crearlos automáticamente cuando lancemos la migración de las tablas a la base de datos.

```
<?=
public function run()
{
    \App\Models\User::factory(10)->create();
}
```

Una vez hecho esto, vamos a crear los usuarios y las tablas en la base de datos. Para ello lanzamos el siguiente comando:

```
sail artisan migrate --seed
```

2.5 CRUD

Ahora que ya tenemos el modelo, es hora de ponerse manos a la obra con el CRUD, así que lo primero que vamos a hacer es crear el controlador con el siguiente comando:

```
sail php artisan make:controller Api/V1/PostController --api --model=Post
```

Al igual que el controlador de autenticación, `PostController` será guardado dentro de la carpeta V1 para poder separar futuras versiones de la `API`. El parámetro `--api` creará automáticamente los métodos del `CRUD` y `--model=Post` le indicará que este controlador trabajará con los datos del modelo `Post`.

Ahora que ya tenemos el controlador, vamos a generar las rutas así que vamos al archivo `routes/api.php` y añadimos la siguiente línea:

```
<?=
//en una sola línea
Route::apiResource('v1/posts',
                   App\Http\Controllers\Api\V1\PostController::class)
                    ->middleware('api');
```

Con esta simple línea crearemos todas las rutas con las que podremos acceder a los cinco métodos de `PostControlle` que podrán realizar las siguientes acciones:

- Mostrar un listado de todos los post con las fotos.
- Mostrar un post en concreto por el id.
- Crearlas.
- Actualizarlas

- Eliminarlas.

Para ver las rutas generadas, podéis lanzar el siguiente comando:

```
sail artisan route:list
```

Ahora tenemos las rutas, pero nos falta controlar que los `endpoints` para la *creación, actualización y borrado* necesiten de un usuario autenticado para poder utilizarlos. Para añadir este control, debemos ir al archivo `app/Http/Controllers/Api/V1/PostController.php` y en él crearemos el constructor de la clase en el que añadiremos el siguiente código:

```
public function __construct()
{
    $this->middleware('auth:api', ['except' => ['index', 'show']]);
}
```

Con esta línea le diremos a Laravel que, a excepción de los métodos `index` y `show`, los demás necesitarán pasar por el middleware `auth:api`, el cual se encargará de *validar* el `token` que pasemos por la cabecera.

2.5.1 Creación

Ahora que ya tenemos las rutas y el controlador es hora de darle vidilla a nuestra `API`. Para ello, lo primero que haremos será habilitar la funcionalidad para añadir nuevos posts para más adelante poder probar los endpoints para listar todos los post y el post por id.

Lo primero que vamos a hacer es crear un objeto `Request`. Esta funcionalidad la utilizaremos para validar los datos que nos envíe el usuario. Para crear la request lanzamos el siguiente comando:

```
sail artisan make:request V1/PostRequest
```

Al lanzar este comando nos creará un archivo dentro de la ruta `app/Http/Requests/V1/PostRequest.php`. Pues bien, lo que vamos a hacer ahora es abrir ese archivo y sustituir su contenido por el siguiente:

```
<?php

namespace App\Http\Requests\V1;

use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Support\Facades\Auth;

class PostRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return Auth::check();
    }

    /**
     * Get the validation rules that apply to the request.
     *
```

```
     * @return array
     */
    public function rules()
    {
        return [
            'title' => 'required|max:70',
            'image' => 'required|image|max:1024',
            'description' => 'required|max:2000',
        ];
    }
}
```

Este archivo tiene dos métodos, el primero, el método `authorize` que sólo **permitirá realizar la petición si el usuario está autenticado**. Para ello utiliza el método `check` de la clase `Auth` que devuelve `true` si el usuario que realiza la petición está autenticado, y si no, devolverá `false`. En caso de no estar autenticado retornará el `error 401 unauthorized`.

El segundo método es el método `rules`. En este método **validaremos** los datos que nos debe enviar el usuario que son el título, la imagen y la descripción. Como podéis ver, retorna un array con formato clave valor en el que cada clave es el campo a enviar y su valor son las distintas reglas que debe cumplir para ser validado separadas por `|`. Si no cumple alguna de estas reglas, Laravel devolverá un error.

Dicho esto vamos a explicar las reglas que vamos a utilizar:

La regla `required` indicará que el campo es obligatorio.

En `title`, `max:70` indicará el número máximo de caracteres que podrá enviar un usuario para asignar un título.

En `image`, añadimos la regla `image` que indica que vamos a enviar un fichero de tipo imagen. En este caso, `max:1024` indicará que el tamaño máximo de una imagen será de *1024 KB*.

En `description` al igual que en title añadimos la regla max esta vez con un límite de *2000 caracteres*.

Aparte de estas hay muchas más reglas, para verlas podéis hacerlo pinchando en el siguiente enlace.

El siguiente paso que vamos a realizar es añadir la funcionalidad para poder *crear post*. Para ello abrimos el archivo `app/Http/Controllers/Api/V1/PostController.php` e importamos las clases `PostRequest` y `Auth`, aparte, añadimos el siguiente código en el método `store` y creamos el método `update` que usaremos para guardar las imágenes.

```
<?php

namespace App\Http\Controllers\Api\V1;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\Request;
use App\Http\Requests\V1\PostRequest;
use Illuminate\Support\Facades\Auth;

class PostController extends Controller
{
...
    /**
```

```
     * Store a newly created resource in storage.
     *
     * @param  App\Http\Requests\V1\PostRequest $request
     * @return \Illuminate\Http\Response
     */
    public function store(PostRequest $request)
    {
        $request->validated();

        $user = Auth::user();

        $post = new Post();
        $post->user()->associate($user);
        $url_image = $this->upload($request->file('image'));
        $post->image = $url_image;
        $post->title = $request->input('title');
        $post->description = $request->input('description');

        $res = $post->save();

        if ($res) {
            return response()->json(['message' => 'Post create succesfully'], 201);
        }
        return response()->json(['message' => 'Error to create post'], 500);
    }

    private function upload($image)
    {
        $path_info = pathinfo($image->getClientOriginalName());
        $post_path = 'images/post';

        $rename = uniqid() . '.' . $path_info['extension'];
        $image->move(public_path() . "/$post_path", $rename);
        return "$post_path/$rename";
    }
...
```

Una vez hecho esto vamos a explicar que es lo que hace el método `store`.

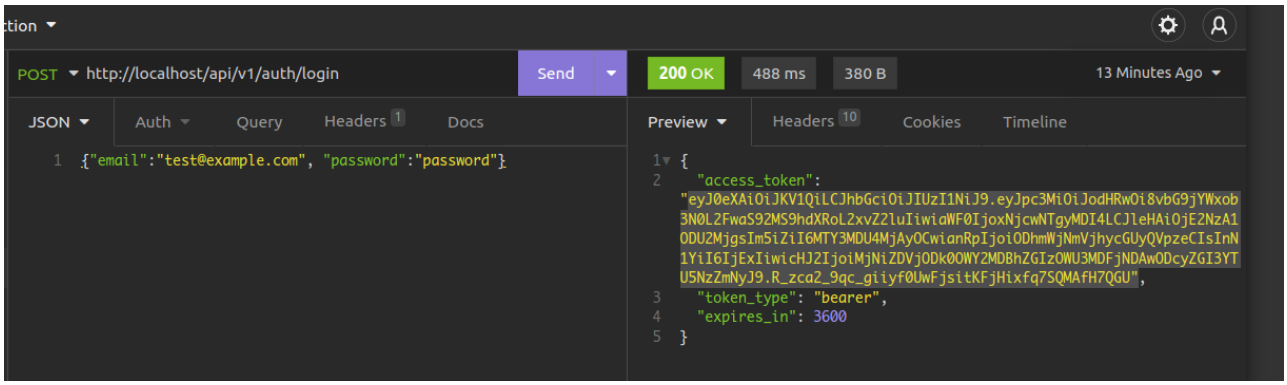
Lo primero que hacemos es validar los datos que nos envía el usuario, para ello lanzamos el método `validated()` del objeto `$request` y este comprobará que se cumplen las reglas que creamos en la clase `PostRequest`.

El siguiente paso es obtener el usuario y crear un objeto de tipo `post` para guardar la información. **Asociamos el usuario al post** y nos valemos del método **upload** para guardar la imagen en la carpeta pública de Laravel y nos retornará su `url`, de esta forma podremos guardar su ruta en la bbdd y visualizarla más adelante.

Añadimos también el título y la descripción y usamos el método `save` del objeto `$post` para guardar los cambios. Si todo ha ido bien retornará `true` y enviaremos al usuario un *mensaje diciendo que todo ha ido bien*, si no, le diremos al usuario que *hemos tenido un error* y nos tocará revisar.

Ahora que ya lo tenemos montado es hora de probarlo. Usaremos Insomnia o Postman, pero podéis utilizar la herramienta que más os guste.

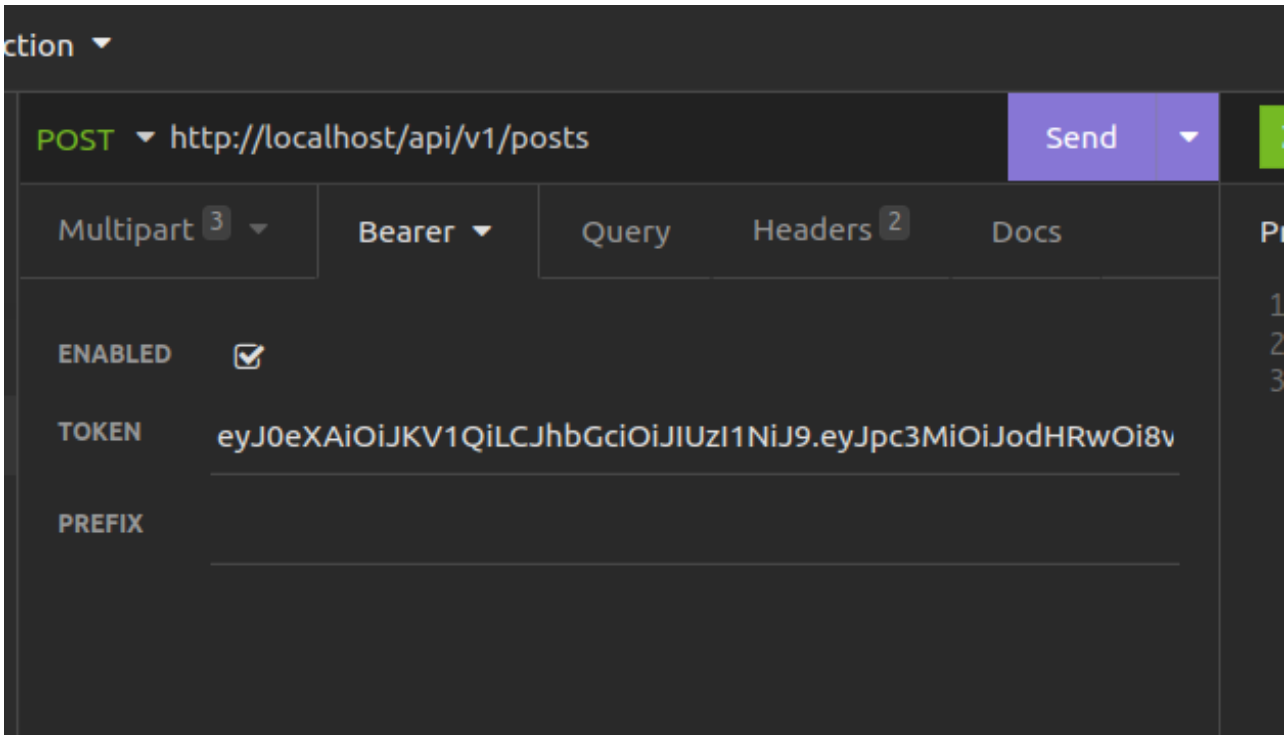
Primero deberemos generar el `token JWT` para poder autenticarnos: En `Insomnia` (o `Postman`), creamos una petición tipo `POST` contra la `url localhost/api/v1/auth/login`



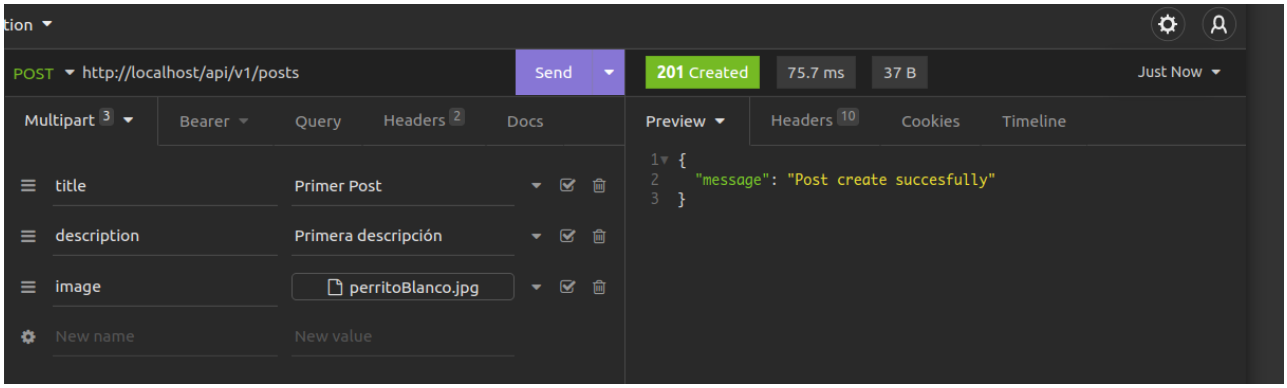
Una vez hecho esto lanzaremos una petición a la siguiente url `http://localhost/api/v1/posts`. Esta petición deberá ser de tipo `POST` y en ella deberemos enviar los parámetros `title`, `image` y `description`. También tenemos que enviar las siguientes cabeceras para que funcione correctamente:

```
Content-type: multipart/form-data;
Authorization: Bearer <my-token>
Accept: application/json
```

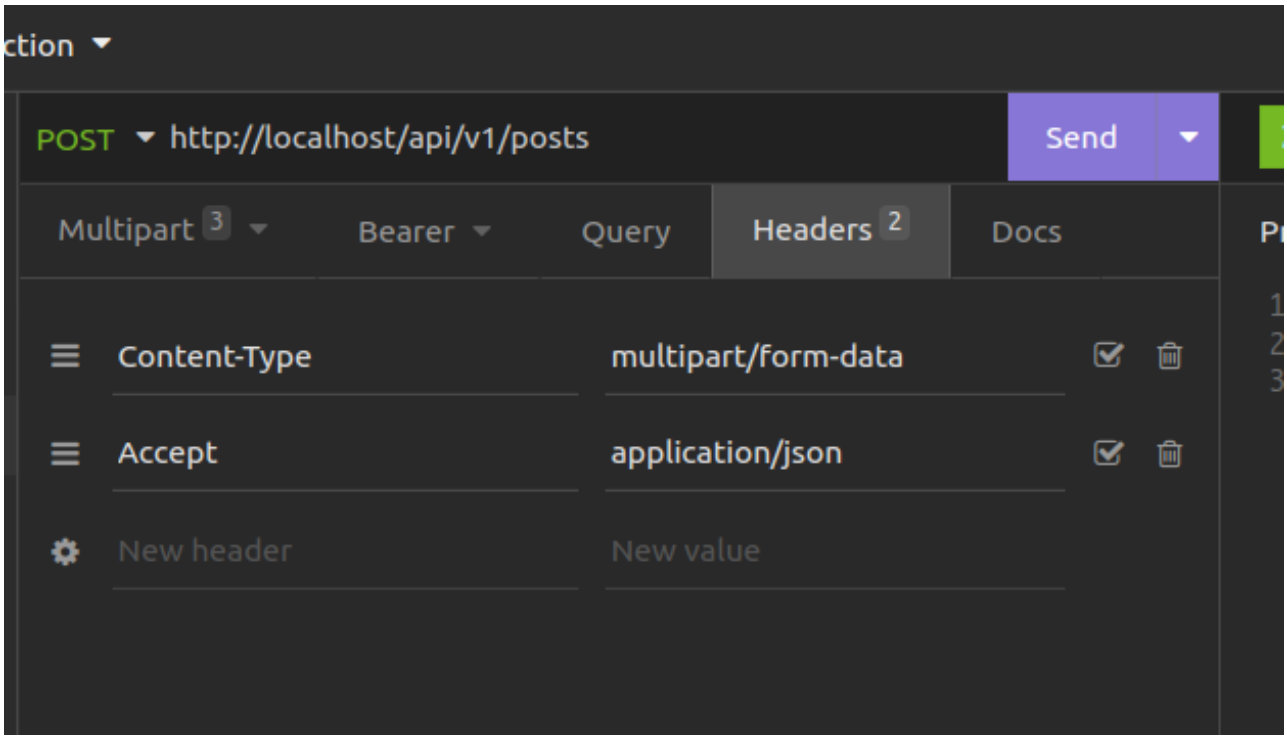
El `Bearer Token` lo podemos poner en la pestaña de `Auth` donde seleccionaremos `Bearer token` y pegamos el `token JWT` obtenido anteriormente.



La creación del post la mandamos en el cuerpo del mensaje —body—.



El resto de cabeceras las añadimos en Headers:



2.5.2 Lectura

Para mostrar datos, Laravel provee una clase intermediaria para poder modificar la respuesta, lo que nos permitiría ocultar `ids`, mostrar lo que queramos... Son los recursos —=Resources=—. Primero la creamos:

```
sail artisan make:resource V1/PostResource
```

Este comando generará un archivo en la ruta `app/Http/Resources/V1/PostResource.php`. Por defecto tiene un método llamado `toArray` que recibe una `Request` (que en este caso sería el `post`) y lo convierte en array, pero nosotros vamos a cambiar levemente el funcionamiento para modificar los datos que queremos mostrar y como mostrarlos. Abrimos el archivo y sustituimos el contenido del método `toArray` por el siguiente:

```php
<?=
public function toArray($request)
{
    return [
        'id' => $this->id,
        'title' => $this->title,
```

```
            'description' => $this->description,
            'photo' => url($this->image),
            'author' => [
                'name' => $this->user->name,
                'email' => $this->user->email,
            ],
            'created_at' => $this->created_at
        ];
    }
```

Cuando creemos un objeto de tipo `PostResource`, este podrá recibir un `post` o una `colección de posts`. Al llamar al campo `toArray`, recorrerá los posts y podremos acceder a sus campos gracias a `$this`. De esta forma hemos podido realizar las siguientes modificaciones:

- Ya no mostramos el campo `updated_at`.
- Hemos renombrado el campo `image` por `photo`, además añadimos la `url` y no sólo donde está almacenada en el proyecto. Esto lo hacemos porque sin ello no podríamos ver la imagen.
- Hemos añadido el `nombre de usuario` y el `email` del autor del `post`.

Ahora que ya tenemos el `Resource` configurado es hora de utilizarlo, para ello abrimos el archivo `PostController`. añadimos la clase `PostResource` y modificamos los métodos `index` y `show` con el siguiente código:

```
<?php

...
use App\Http\Resources\V1\PostResource;

class PostController extends Controller
{
    ...
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        return PostResource::collection(Post::latest()->paginate());
    }
    ...
    public function show(Post $post)
    {
         return new PostResource($post);
    }
    ...
```

El método `index` se encargará de *retornar* todos los `posts` disponibles. Al utilizar el método `collection` de `PostResource` y al devolver los posts páginados con el modelo `Post`, obtendremos un listado de los 15 últimos posts. Además nos mostrará los *links a sucesivas páginas de posts* y una clave `meta` con información como el número total de posts, posts por página, etc. Vamos que nos da mucha información sin hacer prácticamente nada.

Para obtener el *listado de posts* sólo tendremos que lanzar una petición de tipo `GET` a la siguiente url: `localhost/api/`

Este endpoint nos devolverá el siguiente `json` (cambiando mis posts por los vuestros):


```
{
        "data": [
                {
                        "id": 1,
                        "title": "Primer Post",
                        "description": "Primera descripción",
                        "photo": "http:\/\/localhost\/images\/post\/639311daf18ce.jpg"
                        "author": {
                                "name": "Test User",
                                "email": "test@example.com"
                        },
                        "created_at": "2022-12-09T10:45:46.000000Z"
                }
        ],
        "links": {
                "first": "http:\/\/localhost\/api\/v1\/posts?page=1",
                "last": "http:\/\/localhost\/api\/v1\/posts?page=1",
                "prev": null,
                "next": null
        },
        "meta": {
                "current_page": 1,
                "from": 1,
                "last_page": 1,
                "links": [
                        {
                                "url": null,
                                "label": "&laquo; Previous",
                                "active": false
                        },
                        {
                                "url": "http:\/\/localhost\/api\/v1\/posts?page=1",
                                "label": "1",
                                "active": true
                        },
                        {
                                "url": null,
                                "label": "Next &raquo;",
                                "active": false
                        }
                ],
                "path": "http:\/\/localhost\/api\/v1\/posts",
                "per_page": 15,
                "to": 1,
                "total": 1
        }
}
```

El método `show` se encargará de mostrarnos un único post enviando el `id` por la `url`. Este método crea un objeto de tipo `PostResource` y recibe el objeto post en el constructor.

Para realizar la petición debemos lanzar una petición de tipo `GET` a la siguiente url `http://localhost:8000/api/v` sustituyendo el id del post que queráis mostrar por el **1** que es que quiero visualizar yo:

Esta petición devolverá el siguiente resultado:


```
{
        "data": {
                "id": 1,
                "title": "Primer Post",
                "description": "Primera descripción",
                "photo": "http:\/\/localhost\/images\/post\/639311daf18ce.jpg",
                "author": {
                        "name": "Test User",
                        "email": "test@example.com"
                },
                "created_at": "2022-12-09T10:45:46.000000Z"
        }
}
```


2.5.3 Actualización

El siguiente paso es crear la actualización, para ello vamos al archivo `PostController.php`, importamos la clase `Validator` y modificamos el método `update` con el siguiente código:

```
<?=
...
use Illuminate\Support\Facades\Validator;

class PostController extends Controller
{

    public function update(Request $request, Post $post)
    {
        Validator::make($request->all(), [
            'title' => 'max:191',
            'image' => 'image|max:1024',
            'description' => 'max:2000',
        ])->validate();

        if (Auth::id() !== $post->user->id) {
            return response()->json(['message' => 'You don\'t have permissions'], 403)
        }

        if (!empty($request->input('title'))) {
            $post->title = $request->input('title');
        }
        if (!empty($request->input('description'))) {
            $post->description = $request->input('description');
        }
```

```php
        if (!empty($request->file('image'))) {
            $url_image = $this->upload($request->file('image'));
            $post->image = $url_image;
        }

        $res = $post->save();

        if ($res) {
            return response()->json(['message' => 'Post update succesfully']);
        }

        return response()->json(['message' => 'Error to update post'], 500);
    }
```

Una vez añadido el código vamos a explicar que hace:

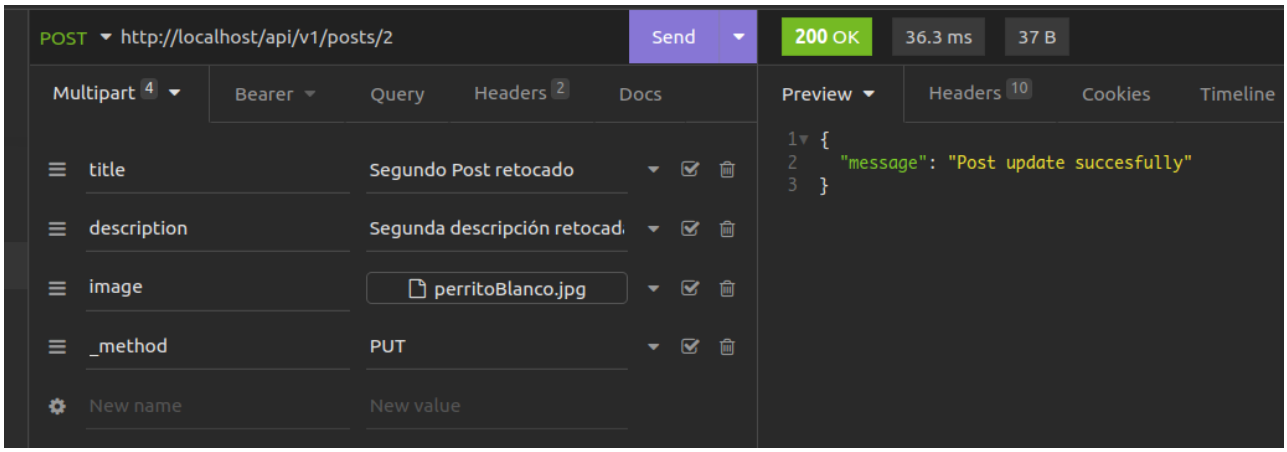
- En este caso, en vez de crear un archivo `request`, se ha añadido la validación desde el propio método gracias a la clase `Validator`. Como veis solo tenemos que añadir las reglas como en el archivo `request` y lanzar el método `validate()`.
- El siguiente paso es verificar que el usuario que realiza la petición es el *propietario del post*. Si no es así *mostraremos un error*.
- Después comprobamos que datos ha enviado el usuario y que sólo modificamos los existentes.
- Por último guardamos y si todo ha ido bien retornamos un mensaje afirmativo.

Importante:

Para lanzar la actualización en Laravel **NO** debemos usar el método `PUT` o `PATCH` en los casos en los que enviemos archivos, ya que `PHP` **NO guarda** la información en la super variable `$_FILES` en una petición `PUT` o `PATCH` y por lo tanto no podremos actualizar este campo si se da el caso. Tendremos que lanzar una petición de tipo `POST` y enviar el parámetro **_method: `PUT`** en la cabecera.

La petición será igual a la que realizamos para pedir un `POST` en el que debemos añadir el `id` del post que queremos actualizar.

Os dejo captura para que lo veáis más claro:



2.5.4 Borrado

Por último vamos a crear la acción de borrado. Volvemos al archivo `PostController.php` y modificamos el método `destroy` por el siguiente código:

```php
<?=
public function destroy(Post $post)
{
    if (Auth::id() !== $post->user_id) {
        return response()->json(['message' => 'You are not the owner of this post'], 4
    }

    // Eliminar la imagen asociada
    $this->deleteImage($post->image);

    $res = $post->delete();
    if ($res) {
        return response()->json(['message' => 'Post deleted successfully'], 200);
    }
    return response()->json(['message' => 'Error to delete post'], 500);
}

private function deleteImage($imagePath)
{
        $fullPath = public_path($imagePath);
        if (file_exists($fullPath)) {
            @unlink($fullPath); // Elimina el archivo
        }
}
```

La función `deleteImage` se encarga de eliminar el archivo de la imagen del disco.

- Obtener la ruta completa: Convierte la ruta relativa de la imagen ($imagePath) a una ruta completa en el sistema de archivos (`$fullPath = public_path($imagePath)`).
- Comprobar si el archivo existe: Verifica si el archivo existe en la ruta especificada (if (file_exists($fullPath))).
- Eliminar el archivo: Si el archivo existe, lo elimina usando la función unlink() de PHP (@unlink($fullPath)). El operador @ suprime cualquier error que pueda ocurrir durante la eliminación.

En este caso solo debemos lanzar el método `delete` y listo, nuestro post habrá sido eliminado.

Para lanzar la petición deberemos utilizar el método `DELETE` y usar la siguiente `url` sustituyendo el número final por el `id` del post que queramos eliminar, ejemplo: `http://localhost:8000/api/v1/posts/1`

2.6 Control de Errores en la API

En este paso, añadiremos un sistema de control de errores a nuestra API para mejorar la respuesta en casos de errores como peticiones inválidas, recursos no encontrados o errores del servidor.

2.7 2.6.1 Middleware de Manejo de Errores

Laravel maneja excepciones mediante la clase `Handler.php`, pero podemos crear un middleware específico para manejar respuestas de error de forma más estructurada.

Ejecutamos el siguiente comando para crear un middleware de manejo de errores:

```
sail artisan make:middleware ApiExceptionHandler
```

Luego editamos `app/Http/Middleware/ApiExceptionHandler.php` y agregamos lo siguiente:

```
namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\JsonResponse;
use Illuminate\Database\Eloquent\ModelNotFoundException;
use Symfony\Component\HttpKernel\Exception\HttpException;
use Throwable;

class ApiExceptionHandler
{
    public function handle($request, Closure $next)
    {
        try {
            return $next($request);
        } catch (ModelNotFoundException $e) {
            return response()->json(['error' => 'Recurso no encontrado'], 404);
        } catch (HttpException $e) {
            return response()->json(['error' => $e->getMessage()], $e->getStatusCode()
        } catch (Throwable $e) {
            return response()->json(['error' => 'Error interno del servidor'], 500);
        }
    }
}
```

Registramos el middleware en `app/Http/Kernel.php` dentro de `$middlewareAliases`:

```
'api.exception' => \App\Http\Middleware\ApiExceptionHandler::class,
```

Ahora podemos aplicar este middleware en nuestras rutas de la API.

2.8 Manejo de Excepciones en los Controladores

En nuestros controladores, podemos lanzar excepciones en caso de errores. Modificamos `PostController.php` para añadir un manejo adecuado en los métodos `show`, `update` y `destroy`.

```
use Illuminate\Http\JsonResponse;
use Illuminate\Database\Eloquent\ModelNotFoundException;

public function show(Post $post): JsonResponse
{
    if (!$post) {
        throw new ModelNotFoundException("Post no encontrado");
    }
    return response()->json(new PostResource($post), 200);
}

public function update(Request $request, Post $post): JsonResponse
{
    if (Auth::id() !== $post->user->id) {
        return response()->json(['message' => 'No tienes permisos'], 403);
    }
    // Actualización del post...
```

```
}

public function destroy(Post $post): JsonResponse
{
    if (!$post) {
        throw new ModelNotFoundException("Post no encontrado");
    }
    $post->delete();
    return response()->json(['message' => 'Post eliminado'], 200);
}
```

2.9 Validaciones Mejoradas en Requests

Modificamos `PostRequest.php` para personalizar los mensajes de error:

```
public function messages()
{
    return [
        'title.required' => 'El título es obligatorio.',
        'title.max' => 'El título no puede superar los 70 caracteres.',
        'image.required' => 'La imagen es obligatoria.',
        'image.image' => 'El archivo debe ser una imagen.',
        'description.required' => 'La descripción es obligatoria.',
    ];
}
```

2.9.1 Middleware no intercepta errores al pedir un post inexistente

Laravel lanza una `ModelNotFoundException` cuando se usa "Route Model Binding" (`Post $post`) y el post no existe. Esto no se capturaba en el middleware por defecto.

- Solución: Modificamos `ApiExceptionHandler.php` para capturar `ModelNotFoundException`:

  ```
  use Illuminate\Database\Eloquent\ModelNotFoundException;

  public function handle($request, Closure $next)
  {
      try {
          return $next($request);
      } catch (ModelNotFoundException $e) {
          return response()->json(['error' => 'Recurso no encontrado'], 404);
      }
      // Otras capturas de errores...
  }
  ```

  Después de estos cambios, la API devolverá respuestas JSON más claras y estructuradas en caso de errores.

2.10 Troubleshooting de Errores Comunes

2.10.1 "Please provide a valid cache path"

Este error ocurre cuando Laravel no puede escribir en la caché debido a un problema con la ruta de almacenamiento.

- Solución: Ejecuta estos comandos para limpiar y regenerar la caché:

```
sail artisan config:clear
sail artisan cache:clear
sail artisan config:cache
```

Si el problema persiste, asegúrate de que la carpeta `storage/framework/cache` existe y tiene los permisos correctos:

```
sail artisan storage:link
chmod -R 775 storage bootstrap/cache
```

3 2.7 Próximos Pasos en la API

Ahora que hemos implementado un control de errores sólido en nuestra API de Laravel 10, hay algunos aspectos adicionales que se pueden mejorar para optimizar su funcionamiento y seguridad.

3.1 2.7.1 Implementación de Logging de Errores

Es recomendable registrar los errores para facilitar la depuración y el monitoreo del sistema. Podemos hacerlo en `Handler.php` utilizando Laravel Logging:

```
use Illuminate\Support\Facades\Log;

public function render($request, Throwable $exception)
{
    Log::error('Error en la API', [
        'exception' => $exception,
        'url' => $request->fullUrl(),
        'user' => auth()->user() ? auth()->user()->id : 'guest'
    ]);

    return parent::render($request, $exception);
}
```

Esto guardará los errores en `storage/logs/laravel.log` para su análisis posterior.

3.2 2.7.2 Configuración de CORS para Integración con Vue.js

Para permitir que el frontend en Vue.js pueda consumir la API sin problemas de seguridad, configuramos CORS en `config/cors.php`:

```
return [
    'paths' => ['api/*', 'sanctum/csrf-cookie'],
    'allowed_methods' => ['*'],
    'allowed_origins' => ['http://localhost:5173'], // Ajustar al dominio del frontend
    'allowed_origins_patterns' => [],
    'allowed_headers' => ['*'],
    'exposed_headers' => [],
    'max_age' => 0,
    'supports_credentials' => true,
];
```

También agregamos el middleware de CORS en `app/Http/Kernel.php`:

```
protected $middlewareGroups = [
    'api' => [
        \Illuminate\Cors\Middleware\HandleCors::class,
        'throttle:60,1',
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
];
```

Esto permitirá que Vue.js pueda interactuar con la API sin bloqueos por política de seguridad del navegador.

3.3 2.7.2b Implementación de Rate Limiting

Para evitar abusos en nuestra API, podemos implementar límites de tasa en `app/Http/Kernel.php`:

```
protected $middlewareGroups = [
    'api' => [
        'throttle:60,1', // 60 solicitudes por minuto
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
];
```

Esto limitará las solicitudes de un mismo usuario en un período de tiempo.

3.4 2.7.3 Documentación con OpenAPI (Swagger)

Para mejorar la usabilidad de nuestra API, podemos generar documentación automática con Swagger. Instalamos el paquete:

```
composer require darkaonline/l5-swagger
```

Y generamos la documentación con:

```
php artisan l5-swagger:generate
```

Ahora los usuarios podrán acceder a `http://localhost/api/documentation` para consultar la API.

3.5 2.7.4 Pruebas Unitarias y de Integración

Añadimos pruebas para asegurar que los errores se manejan correctamente. Creamos un test:

```
php artisan make:test ApiErrorHandlingTest
```

Ejemplo de prueba en `tests/Feature/ApiErrorHandlingTest.php`:

```
public function test_not_found_error()
{
    $response = $this->getJson('/api/v1/posts/9999');
    $response->assertStatus(404)->assertJson(['error' => 'Recurso no encontrado']);
}
```

3.6 2.7.5 Seguridad Adicional: Validación de Payloads con JSON Schema

Podemos asegurar que las solicitudes cumplen con un esquema predefinido utilizando el paquete `justinrainbow/jso`

```
composer require justinrainbow/json-schema
```

Y validamos en nuestros controladores:

```
use JsonSchema\Validator;

$validator = new Validator;
$validator->validate($request->all(),
 (object)['type' => 'object', 'required' => ['title', 'description']]);
if (!$validator->isValid()) {
    return response()->json(['error' => 'Payload inválido'], 400);
}
```

3.7 2.7.6 Implementación de Roles y Permisos

Para gestionar permisos en la API, podemos utilizar Laravel **Gates** o el paquete `spatie/laravel-permission`.

3.7.1 Instalación del paquete de roles y permisos

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

Publicamos la configuración y ejecutamos las migraciones:

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

3.7.2 Definir Roles y Permisos

En `App/Models/User.php`, añadimos:

```
use Spatie\Permission\Traits\HasRoles;

class User extends Authenticatable
{
    use HasRoles;
}
```

En un Seeder, creamos roles y permisos:

```
use Spatie\Permission\Models\Role;
use Spatie\Permission\Models\Permission;

Role::create(['name' => 'admin']);
Role::create(['name' => 'editor']);
Role::create(['name' => 'user']);

Permission::create(['name' => 'create post']);
Permission::create(['name' => 'edit post']);
Permission::create(['name' => 'delete post']);

$admin = Role::findByName('admin');
$admin->givePermissionTo(['create post', 'edit post', 'delete post']);
```

3.7.3 Aplicar Permisos en Controladores

Podemos restringir el acceso con middleware en `routes/api.php`:

```
Route::middleware(['auth:sanctum', 'role:admin'])->group(function () {
  Route::post('/posts', [PostController::class, 'store']);
```

```
  Route::delete('/posts/{post}', [PostController::class, 'destroy']);
});
```

O dentro del controlador:

```
public function store(Request $request)
{
    if (!auth()->user()->can('create post')) {
        return response()->json(['error' =>
                            'No tienes permisos para crear posts'], 403);
    }
    // Lógica para crear post...
}
```

Con esto, nuestra API ahora implementa un control de roles y permisos basado en Laravel Permission.

3.7.4 Ocultar Posts

Para permitir que administradores y editores oculten posts, agregamos un nuevo campo `hidden` en la migración de posts:

```
Schema::table('posts', function (Blueprint $table) {
    $table->boolean('hidden')->default(false);
});
```

Modificamos `PostController.php` para permitir ocultar un post:

```
public function hide(Post $post)
{
    $user = auth()->user();
//¿Cómo podrías hacer que el propio creador pueda ocultar el post?
    if ($user->hasRole('admin') || $user->hasRole('editor')) {
        $post->hidden = true;
        $post->save();

        // Notificar al autor del post
        $post->user->notify(new PostHiddenNotification($post));

        return response()->json(['message' => 'El post ha sido ocultado'], 200);
    }

    return response()->json(['error' =>
 'No tienes permisos para ocultar este post'], 403);
}
```

Finalmente, creamos la notificación `PostHiddenNotification` para avisar al autor:

```
php artisan make:notification PostHiddenNotification
```

Editamos `app/Notifications/PostHiddenNotification.php`:

```
use Illuminate\Notifications\Notification;
use Illuminate\Notifications\Messages\MailMessage;

class PostHiddenNotification extends Notification
{
```

```
    protected $post;

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

    public function via($notifiable)
    {
        return ['mail'];
    }

    public function toMail($notifiable)
    {
        return (new MailMessage)
            ->subject('Tu post ha sido ocultado')
            ->line('Tu post ha sido ocultado por un moderador debido a contenido inapr
            ->action('Ver detalles', url('/posts/' . $this->post->id));
    }
}
```

Con esto, los administradores y editores podrán ocultar posts inapropiados y notificar al autor.

3.7.5 Conclusiones

Y listo, ya tenemos nuestra API completamente funcional con Laravel 10. Como podéis ver Laravel se encarga de realizar mucho trabajo por nosotros lo que hará que podamos ahorrar mucho tiempo a la hora de montar una API.

3.7.6 Por hacer

- Si hay tiempo veremos una autenticación mejor que la de JWT, con Laravel Sanctum.
- Poner una API completa con Flask, para que la tengáis de referencia los que queráis saber algo de Flask.
- Conectar el servidor con el cliente. Esto ya es cosa de... "Cliente"

https://notasweb.me/entrada/crear-un-api-rest-en-laravel

https://cosasdedevs.com/posts/crud-api-laravel-8-parte-1-modelos-creacion/

https://www.iankumu.com/blog/laravel-rest-api/ Aut. con JWT https://blog.logrocket.com/implementing-jwt-authentication-laravel-9/

https://www.linkedin.com/pulse/rest-api-laravel-10-using-jwt-token-muhammad-babar-gvq

4 Apendice I: Recomponer vendor

```
docker run --rm \
    -u "$(id -u):$(id -g)" \
    -v "$(pwd):/var/www/html" \
    -w /var/www/html \
    laravelsail/php81-composer:latest \
```

```
composer install --ignore-platform-reqs
```

Footnotes: [1] Cuando clonamos desde un repositorio el «secret» no está en el repositorio, por lo que hay que volver a lanzar el comando.