CREACIÓN DE API REST EN LARAVEL

DESARROLLO DE APLICACIONES WEB

Asignatura: Desarrollo de aplicaciones en entorno servidor

Curso: 2023/2024

Autor: Cristina Delgado Muñoz

Pareja: José Miguel Martín Rojas

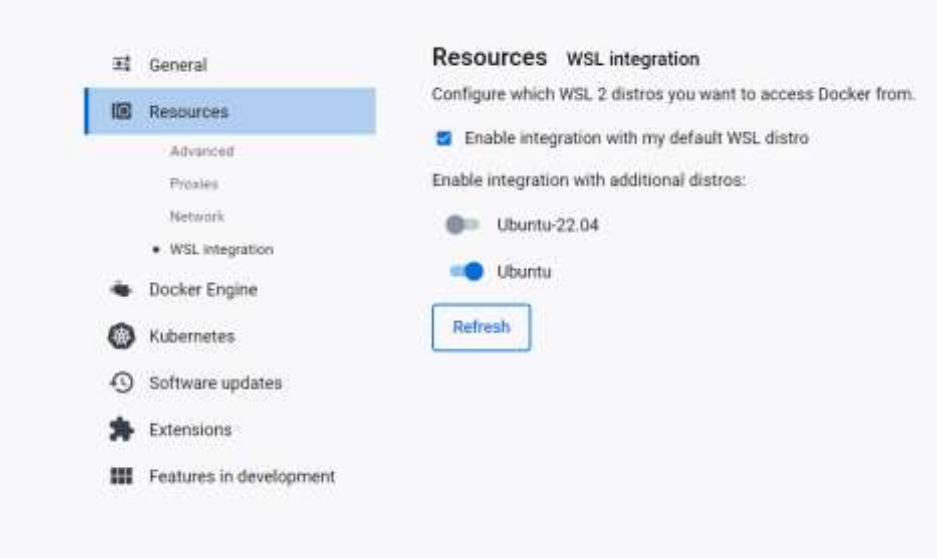
Tabla de contenido

INTRODUCCIÓN

En este proyecto crearemos una apiRest en laravel que nos permitirá acceder por medio del uso de endpoints a las tablas tareas y etiquetas. Para ello, necesitaremos, además, utilizar la tabla users para realizar la autentificación que nos permitirá realizar las consultas.

1. Preparación del entorno

En Docker desktop, vamos a configuración > resources > WSL integration. Una vez ahí, comprobamos que Ubuntu esté encendido.



1.1. Crear el proyecto:

Accedemos a la consola de Ubuntu y vamos al directorio en el que queremos crear el proyecto

```
cristina@Cristinalaptop:~$ cd /mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www$
```

Hacemos composer install para instalar composer, que nos servirá para crear los proyectos de laravel. En mi caso, ya estaba instalado. Una vez hecho esto, escribimos el siguiente comando:

“composer create-project --prefer-dist laravel/laravel <nombreDelProyecto>”. En mi caso, el nombre del proyecto es “apiTarea”

Una vez termina de crearse el proyecto, accedemos al directorio de la api y ejecutamos “php artisan sail:install”. Elegimos el servicio de base de datos, en mi caso mysql.

Instalamos npm con el comando npm install.

Una vez instalado, ejecutamos el comando “alias sail=./vendor/bin/sail” para trabajar más fácilmente con el sail que hemos instalado, y “sail up -d” para ejecutar el proyecto. Es importante añadir “-d” para poder seguir ejecutando comandos en la terminal Ubuntu sin tener que salir del proyecto.

Si abrimos con visual studio la carpeta apiTarea, este es su contenido:


APITAREA
app
bootstrap
config
database
node_modules
public
resources
routes
storage
tests
vendor
.editorconfig
.env
.env.example
.gitattributes
.gitignore
artisan
composer.json
composer.lock
docker-compose.yml
package-lock.json
package.json
phpunit.xml
README.md
vite.config.js

2. Tareas y etiquetas

2.1 Crear migraciones

Las migraciones son una característica de laravel que permite crear estructuras de datos para utilizar la base de datos por medio de php.

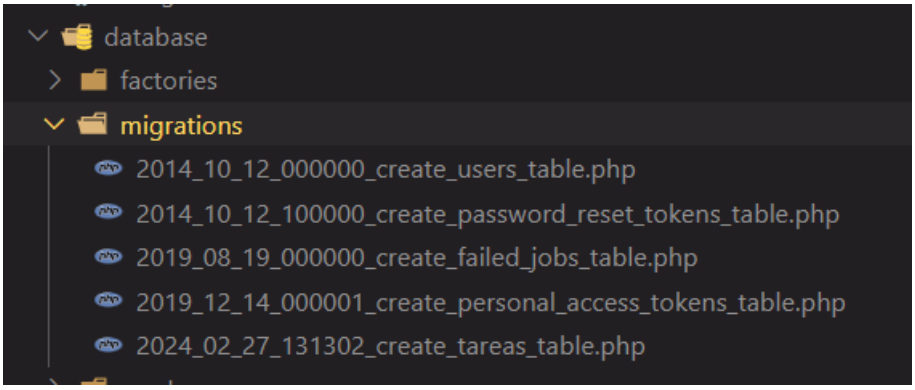
En esta tarea, vamos a tener una tabla “tareas”, otra “etiquetas” y una última que relacione ambas tablas con una relación muchos a muchos (belongs to many, lo veremos más adelante). La que relacione ambas tareas se llamará tareas_etiquetas.

Comando para crear una migración:
sail artisan make:migration create_<nombreTabla>_table

2.1.1. Migración de la tabla “tareas”

Ejecutamos “sail artisan make:migration create_tareas_table”.

Podemos ver que se ha creado un archivo dentro de database > migrations cuyo nombre contiene la fecha de creación del archivo seguida de “create_tareas_table.php”.



En este archivo, estableceremos la estructura de la tabla tareas. En la función “up”, crearemos las columnas de la tabla como se ve a continuación:

```php
public function up(): void
{
    Schema::create('tareas', function (Blueprint $table) {
        $table->id();
        $table->string("titulo", 20);
        $table->string("descripcion", 200);
        $table->timestamps();
    });
}
```

También se creará la función “down”, que servirá para revertir la migración de esta tabla si es necesario.

```php
public function down(): void
{
    Schema::dropIfExists('tareas');
}
};
```

Una vez hecho esto, podemos ejecutar en la consola Ubuntu el comando “sail artisan migrate” para ejecutar la migración.

2.1.2 Migración de la tabla “etiquetas”

Ejecutamos “sail artisan make:migration create_etiquetas_table”

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea$
sail artisan make:migration create_etiquetas_table

  INFO  Migration [database/migrations/2024_02_27_132415_create_etiquetas_table.php] cre
ated successfully.
```

Vamos al archivo que se acaba de crear con el comando y hacemos los cambios en la función up:

```php
public function up(): void
{
    Schema::create('etiquetas', function (Blueprint $table) {
        $table->id();
        $table->string("nombre", 20);
        $table->timestamps();
    });
}
```

Ejecutamos “sail artisan migrate”.

```
ristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea$ s
ail artisan migrate

  INFO  Running migrations.

  2024_02_27_132415_create_etiquetas_table ..... 101ms DONE
```

2.1.3 Migración de la tabla “tareas_etiquetas”

Como última migración del proyecto, ejecutamos “sail artisan make:migration create_tareas_etiquetas_table” para crear la tabla que une a las dos anteriores.

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea$
 sail artisan make:migration create_tareas_etiquetas_table

  INFO  Migration [database/migrations/2024_02_27_132938_create_tareas_etiquetas_table.
php] created successfully.
```

En este caso, los campos que aparecerán en la función up, las columnas de la tabla, serán claves foráneas, por lo que la función up será así:

```php
public function up(): void
{
    Schema::create('tareas_etiquetas', function (Blueprint $table) {
        $table->id();
        $table->foreignId("tareas_id")->constrained()->onDelete('cascade');
        $table->foreignId("etiquetas_id")->constrained()->onDelete('cascade');
        $table->timestamps();
    });
}
```

Por último, ejecutamos “sail artisan migrate”.

```
ap/desktop/2DAW/DWEServidor/docker/www/apiTarea$ sail artisan migrate

  INFO  Running migrations.

  2024_02_27_132938_create_tareas_etiquetas_table .....
..... 421ms DONE
```

2.2. Crear modelos

Los modelos en laravel son clases de php que simbolizan una tabla de la base de datos. Sirven para interactuar con la base de datos, y son usados por los seeders para insertar información en las tablas. Sólo será necesario crear un modelo Tarea y un modelo Etiqueta.

Para crear un modelo, ejecutamos el siguiente comando:
sail artisan make:model <nombreModelo> -cr

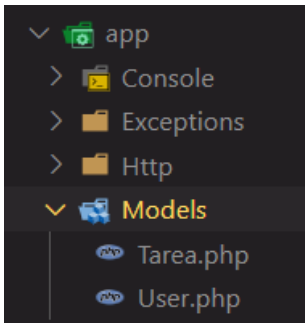
2.2.1. Modelo de Tarea

Para crear el modelo de tarea, ejecutamos el comando sail artisan make:model Tarea -cr

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea$
 sail artisan make:model Tarea -cr

  INFO  Model [app/Models/Tarea.php] created successfully.

  INFO  Controller [app/Http/Controllers/TareaController.php] created successfully.
```

Como podemos ver, además del modelo Tarea se ha creado un controlador.

Encontraremos el modelo de Tarea dentro de app > Models.



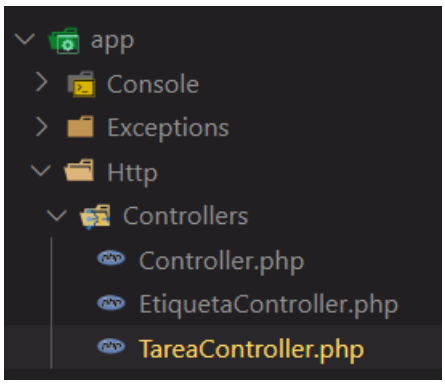
En el archivo del modelo, crearemos la clase tarea. En mi caso, he tratado “nombre” y “descripción” como columnas fillable, es decir, columnas en las que podemos insertar información. Las columnas “created_at” y “updated_at” serán hidden porque no necesitaremos establecer los valores manualmente.

Las etiquetas que están relacionadas con las tareas que insertemos se buscarán con la función etiquetas. En ella, establecemos la relación mencionada antes, belongs to many, que implica que una tarea puede tener más de una etiqueta y una etiqueta puede pertenecer a más de una tarea. En la relación BelongsToMany, estableceremos una unión con la clase Etiqueta por medio de tareas_id y etiquetas_id, que son las claves foráneas que se encuentran en la tabla “tareas_etiquetas” y que establecimos en la migración.

```php
class Tarea extends Model
{
    use HasFactory;

    protected $fillable = ["nombre", "descripcion"];
    protected $hidden = ["created_at", "updated_at"];

    public function etiquetas(): BelongsToMany
    {
        return $this->BelongsToMany(Etiqueta::class, 'tareas_etiquetas', 'tareas_id', 'etiquetas_id');
    }
}
```

Si no hemos creado aún el modelo Etiqueta, aparecerá un error indicando que no se ha podido encontrar la clase Etiqueta.

Como mencioné antes, cuando creamos el modelo también se crea un controlador. Este se puede encontrar en app > Http > Controllers. En este controlador estarán las funciones con las que podemos realizar las operaciones CRUD (creación, lectura, actualización y borrado de datos).



Por ahora no utilizaremos los controladores, ya que necesitamos crear los requests y resources primero.

2.2.2. Modelo de Etiqueta

Para crear el modelo de etiqueta, ejecutamos el comando "sail artisan make:model Etiqueta -cr". De nuevo, podemos ver que se ha creado un modelo y un controlador.

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea$
 s
ail artisan make:model Etiqueta -cr

  INFO  Model [app/Models/Etiqueta.php] created successfully.

  INFO  Controller [app/Http/Controllers/EtiquetaController.php] created successfully.
```

En el modelo, crearemos una clase muy similar a la del modelo de Tarea, en la que también estableceremos una relación belongs to many.

```php
class Etiqueta extends Model
{
    use HasFactory;

    protected $fillable = ["nombre"];
    protected $hidden = ["created_at", "updated_at"];

    public function tareas(): BelongsToMany
    {
        return $this->BelongsToMany(Tarea::class, 'tareas_etiquetas', 'etiquetas_id', 'tareas_id');
    }
}
```

En este caso, se debe poner primero “etiquetas_id” para que funcione adecuadamente.

2.3. Crear seeders

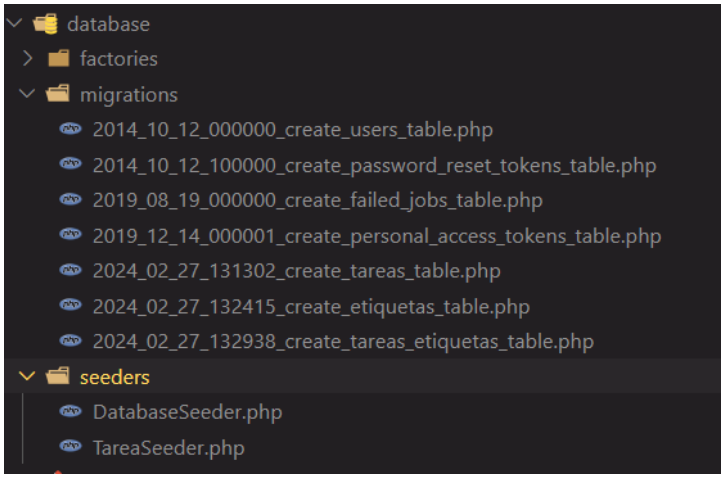
Los seeders en laravel nos permiten insertar datos en las tablas creadas con migration. Vamos a crear seeders para las tres migraciones creadas en el apartado enterior.

2.3.1 Seeder de tarea

Ejecutamos el comando “sail artisan make:seeder TareaSeeder”.

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea$
 sail artisan make:seeder TareaSeeder

  INFO  Seeder [database/seeders/TareaSeeder.php] created successfully.
```

Encontraremos el archivo que se ha creado con este comando en la carpeta database > seeders



Para que el seeder funcione adecuadamente, debemos importar la clase DB al principio del archivo.

```php
namespace Database\Seeders;

use Illuminate\Support\Facades\DB;
use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
```

A continuación, podemos utilizar el método run() para crear tareas. Voy a crear dos tareas de ejemplo:

```php
public function run(): void
{
    DB::table('tareas')->insert(
        [
            "titulo" => "Limpiar",
            "descripcion" => "Fregar el salón y recoger la cocina."
        ]
    );

    DB::table('tareas')->insert(
        [
            "titulo" => "Programar",
            "descripcion" => "Programar una api rest en Laravel."
        ]
    );
}
```

2.3.2. Seeder de Etiqueta

Ejecutamos el comando “sail artisan make:seeder EtiquetaSeeder”.

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea
$ sail artisan make:seeder EtiquetaSeeder

  INFO  Seeder [database/seeders/EtiquetaSeeder.php] created successfully.
```

Al igual que en el seeder de tareas, importamos la clase DB.

```php
namespace Database\Seeders;

use Illuminate\Support\Facades\DB;
use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
```

En el seeder, podemos crear también unas cuantas etiquetas:

```php
public function run(): void
{
    DB::table('etiquetas')->insert(
        [
            "nombre" => "Estilo de vida"
        ]
    );

    DB::table('etiquetas')->insert(
        [
            "nombre" => "Estudios"
        ]
    );
}
```

2.3.3. Seeder de Tarea Etiqueta

Ejecutamos el comando para crear el seeder:

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea$ s
ail artisan make:seeder TareaEtiquetaSeeder

  INFO  Seeder [database/seeders/TareaEtiquetaSeeder.php] created successfully.
```

En el archivo que se ha creado, creamos las relaciones entre las tablas. En este caso, he utilizado el nombre de las tareas y etiquetas para encontrarlas, pero generalmente utilizaremos el id cuando realicemos los endpoints.

```php
public function run(): void
{
    //TAREAS
    $tareaLimpiar = Tarea::where('titulo', 'Limpiar')->first();
    $tareaProgramar= Tarea::where('titulo', 'Programar')->first();
    //ETIQUETAS
    $etiquetaEstiloVida = Etiqueta::where('nombre', 'Estilo de vida')->first();
    $etiquetaEstudio = Etiqueta::where('nombre', 'Estudios')->first();

    $tareaLimpiar->etiquetas()->attach($etiquetaEstiloVida);
    $tareaProgramar->etiquetas()->attach($etiquetaEstiloVida);
    $tareaProgramar->etiquetas()->attach($etiquetaEstudio);
}
```

Cabe destacar que en este caso he utilizado la función first para obtener el primer resultado para hacer un select where utilizando el nombre. Podemos también utilizar firstOrNew si queremos asegurarnos de que si la tarea o la etiqueta no existen se creen, pero he decidido utilizar first.

Es importante importar los modelos de Etiqueta y Tarea al principio del fichero TareaEtiquetaSeeder si queremos que este seeder se ejecute adecuadamente.

```php
namespace Database\Seeders;
use App\Models\Etiqueta;
use App\Models\Tarea;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
```

2.3.4. Ejecución de los seeders

Antes de ejecutar el comando “sail artisan db:seed” para que se guarden los elementos que hemos creado en la base de datos, debemos llamar a los seeders desde el archivo DatabaseSeeder.php.

```php
0 references | 0 overrides | Codiumate: Options | Test this method
public function run(): void
{
    // Llama a los seeders de etiquetas y tareas
    $this->call(EtiquetaSeeder::class);
    $this->call(TareaSeeder::class);

    // Llama al seeder de TareaEtiqueta
    $this->call(TareaEtiquetaSeeder::class);
}
}
```

El último seeder al que llamaremos será TareaEtiqueta ya que necesitamos crear las ocurrencias de tarea y etiqueta antes de crear la relación entre ellas.

Tras ejecutar sail artisan db:seed, los datos se añadirán a la base de datos.

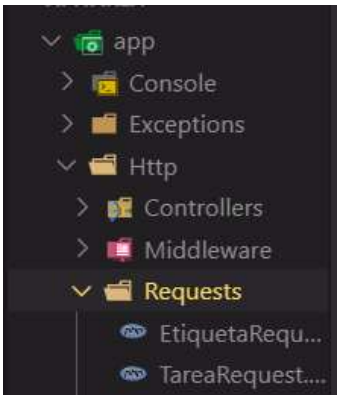
2.4. Crear requests

Los requests, peticiones en español, nos permiten “crear una plantilla” que especifique con qué formato queremos que se devuelvan los datos de una tabla.

2.4.1. Request de Tarea

Para crear el Request de Tarea, realizamos el comando “sail artisan make:request TareaRequest”.

Una vez creado, lo podemos encontrar dentro de app > Http > Requests



En la función authorize, normalmente se comprueba si el usuario va a poder o no acceder a ciertos datos de la tabla. Como en este caso no vamos a tener en cuenta la autorización del usuario aquí, sólo en el login y logout, vamos a hacer que devuelva “true” para que podamos acceder por medio de endpoints. Haremos lo mismo en la tabla etiquetas más adelante.

```
0 references | 0 overrides | Codiumate: Options | Test this meth
public function authorize(): bool
{
    return true;
}
```

Por otro lado, en la función “rules”, normas, estableceremos las reglas relacionadas con la inserción o actualización de cambios en la tabla. He establecido el título como required, es decir, obligatorio. Por otro lado, la descripción podrá estar vacía. También se pueden establecer el máximo y mínimo de caracteres.

```
public function rules(): array
{
    return [
        'titulo' => 'required|max:20|min:3',
        'descripcion' => 'nullable|max:200|min:3',
    ];
}
```

2.4.2. Request de etiqueta

Creamos el Request de Etiqueta con el comando “sail artisan make:request EtiquetaRequest”.

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea$
sail artisan make:request EtiquetaRequest

  INFO  Request [app/Http/Requests/EtiquetaRequest.php] created successfully.
```

Recordamos cambiar el return de authorize de false a true, y en la función rules, he decidido hacer que el título sea required y de entre 3 y 20 caracteres.

```php
public function rules(): array
{
    return [
        'nombre' => 'required|max:20|min:3',
    ];
}
```

2.5. Crear resources

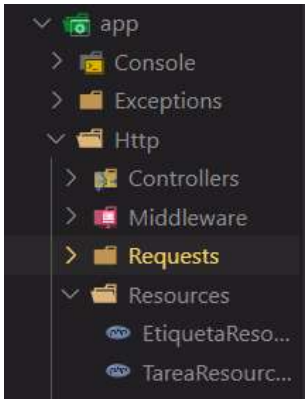
Los resources convierten los datos de la tabla en un array en formato json, que hace más fácil el tratamiento de los datos desde los endpoints.

2.5.1. Resource de Tarea

Para crear el Resource de Tarea, realizamos el comando “sail artisan make:resource TareaResource”

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea$
sail artisan make:resource TareaResource

  INFO  Resource [app/Http/Resources/TareaResource.php] created successfully.
```

Una vez creado, lo podemos encontrar dentro de app > Http > Resources



En la función toArray, devolvemos los datos de una fila de la tabla como un array con formato json.

```php
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'titulo' => $this->titulo,
        'descripcion' => $this->descripcion,
        'etiquetas' => $this->etiquetas
    ];
}
```

2.5.2. Resource de etiqueta

Creamos el Resource de Etiqueta con el comando “sail artisan make:resource EtiquetaResource”.

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea$
sail artisan make:resource EtiquetaResource

  INFO  Resource [app/Http/Resources/EtiquetaResource.php] created successfully.
```

En la función toArray, hacemos lo mismo que en el apartado anterior:

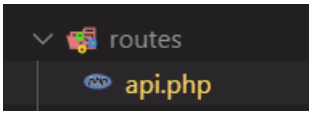
```php
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'nombre' => $this->nombre,
        'tareas' => $this->tareas
    ];
}
```

2.6. Modificar los controladores

Los controladores, controllers, sirven para especificar cuáles serán las acciones que se llevarán a cabo al realizar peticiones por medio de get, post, put y delete.

Ahora que los resources y requests están preparados, podemos realizar cambios en los controllers.

Sin embargo, antes de realizarlos, podemos hacer las llamadas a los endpoints desde api.php. Este documento se encuentra en la carpeta routes.



Al principio del fichero, importamos los controladores de Tarea y Etiqueta, ya que las funciones a las que llamaremos desde api.php se encuentran en esos ficheros.

```php
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\EtiquetaController;
use App\Http\Controllers\TareaController;
```

Con route::resource no será necesario llamar a las funciones de los controladores individualmente.

```php
Route::resource('/tareas',TareaController::class);
Route::resource('/etiquetas',EtiquetaController::class);
```

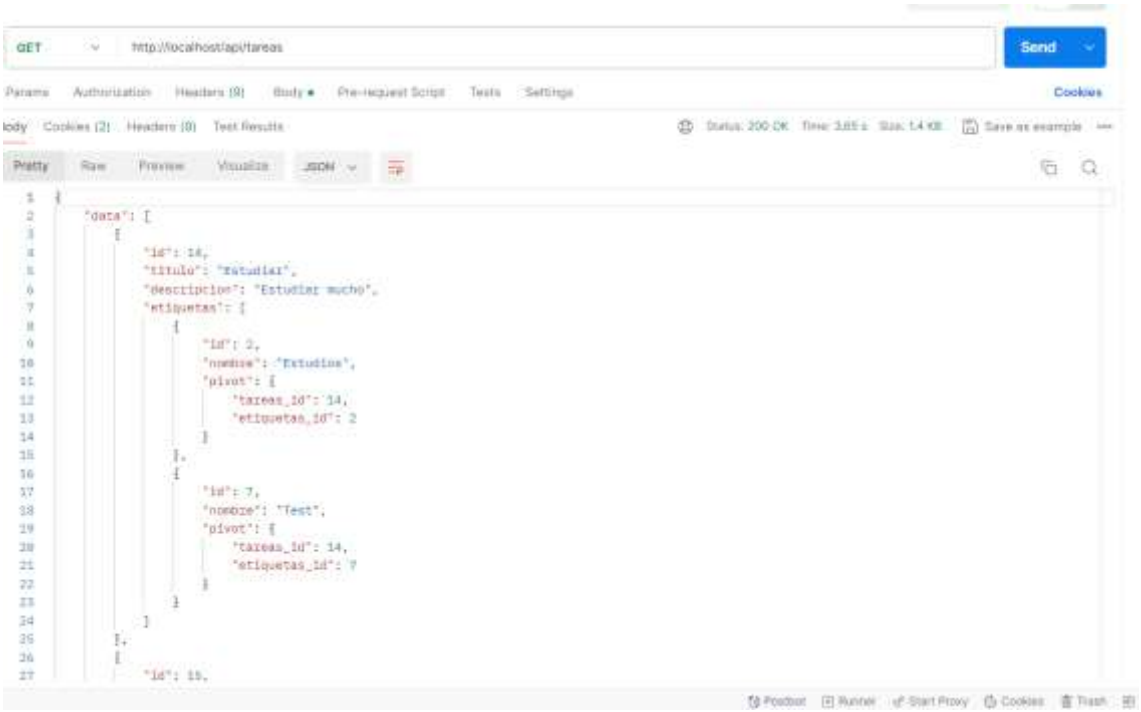
2.6.1. Controller de Tarea

- Función index:

La función index es la que será llamada cuando hagamos una petición get a /tareas. En esta función, estableceremos los pasos necesarios para que se devuelvan todos los datos de las tareas existentes.

```php
public function index()
{
    $tareas = Tarea::all();
    return TareaResource::collection($tareas);
}
```

Al realizar la búsqueda en postman, este es el resultado:



- Función store:

La función store es la que será llamada cuando hagamos una petición post a /tareas. En esta función, estableceremos los pasos necesarios para que se cree una tarea nueva.

```php
public function store(TareaRequest $request)
{
    $tareaNueva = new Tarea();
    $tareaNueva->titulo = $request->titulo;
    $tareaNueva->descripcion = $request->descripcion;

    $tareaNueva->save();

    $etiquetas = $request->etiquetas;
    $tareaId = $tareaNueva->id;
    $tareaNueva->etiquetas()->attach($etiquetas, ['tareas_id' => $tareaId]);

    return new TareaResource($tareaNueva);
}
```

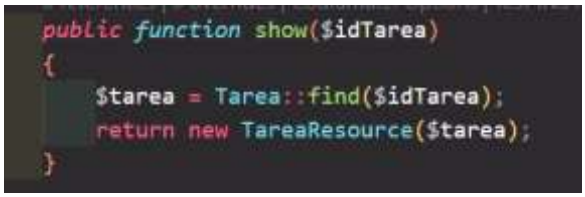
Para comprender esta función, es importante destacar que para que se puedan asociar etiquetas a esta nueva tarea, debemos recoger el id de la tarea. Utilizamos $tareaNueva->save() para guardar la tarea en la tabla antes de pedir su id.

En postman, ponemos los datos de la nueva tarea en Body > raw:

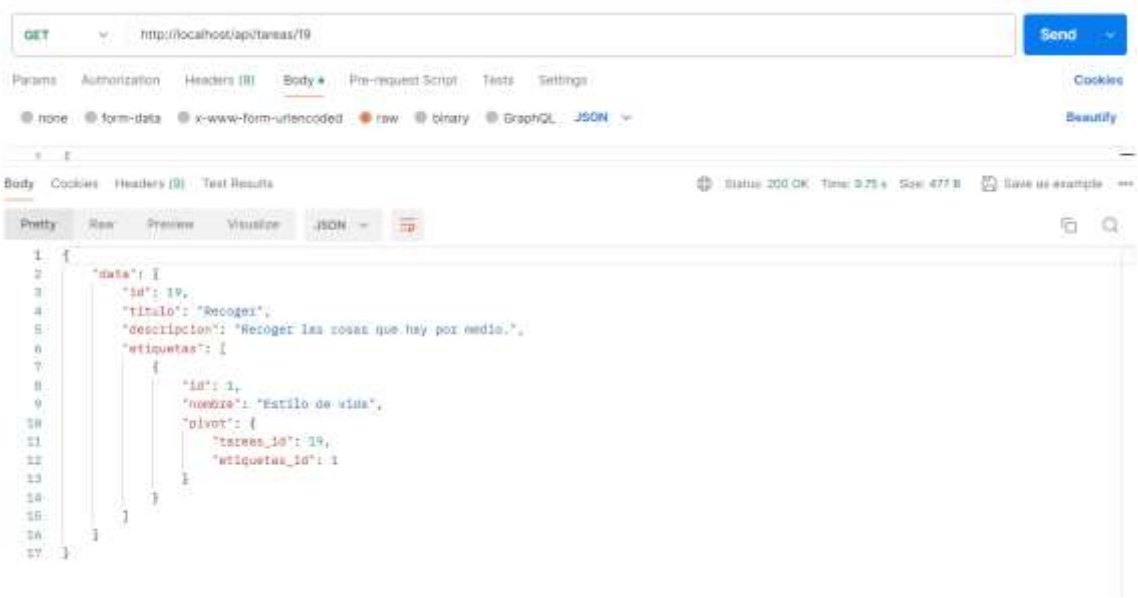
Resultado:

- Función show:

La función show actúa como la función index, pero sólo muestra los datos de una tarea. Para llamarla, es necesario realizar un get en /tareas/<idTarea>



Este es el resultado de consultar el endpoint:



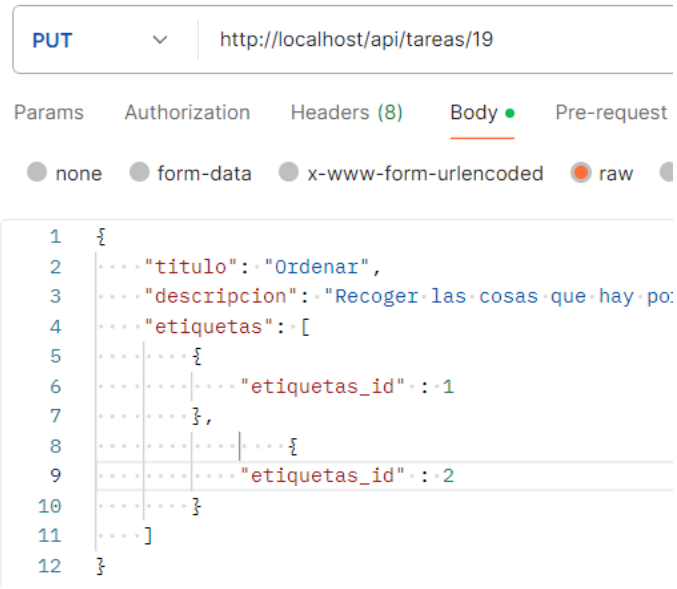
- Función update:

Como su nombre indica, la función update actualiza los datos de una tarea en concreto. Esta función se llama al realizar una consulta put especificando el id de la tarea: /tareas/<idTarea>

```php
public function update(TareaRequest $request, $idTarea)
{
    $tareaModif = Tarea::find($idTarea);
    $tareaModif->titulo = $request->titulo;
    $tareaModif->descripcion = $request->descripcion;

    $tareaModif->etiquetas()->detach();
    $tareaModif->etiquetas()->attach($request->etiquetas);

    $tareaModif->save();

    return new TareaResource($tareaModif);
}
```

En esta función, hemos buscado la tarea por su id, hemos modificado su título y descripción y hemos cambiado el anterior array de etiquetas por uno nuevo obtenido en el request.

Esta es la consulta que debemos realizar en el postman:

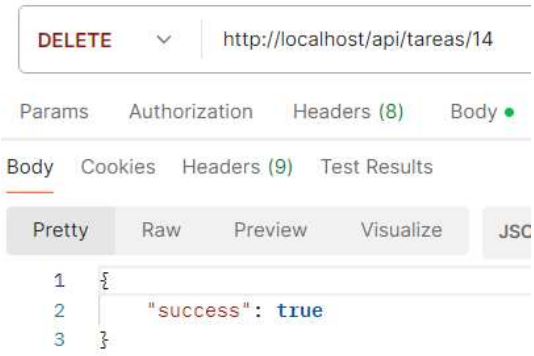


Este es el resultado, cambiando el título por uno nuevo y añadiendo una nueva etiqueta:

- Función destroy:

La función destroy borra una tarea concreta. Se llama a esta función con el endpoint delete /tareas/<idTarea>

Resultado:



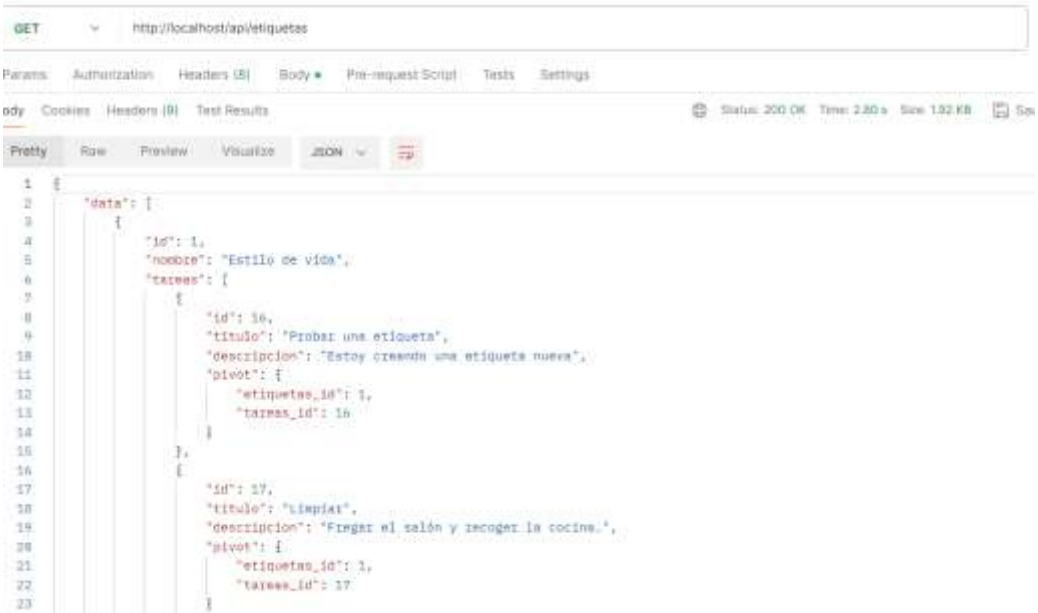
2.6.2. Controller de Etiqueta

- Función index:

La función index es la que será llamada cuando hagamos una petición get a /etiquetas. En esta función, estableceremos los pasos necesarios para que se devuelvan todos los datos de las etiquetas existentes.

```php
public function index()
{
    $etiquetas = Etiqueta::all();
    return EtiquetaResource::collection($etiquetas);
}
```

Al realizar la búsqueda en postman, este es el resultado:



- Función store:

La función store es la que será llamada cuando hagamos una petición post a /etiquetas. En esta función, estableceremos los pasos necesarios para que se cree una tarea nueva.

```php
public function store(EtiquetaRequest $request)
{
    $etiquetaNueva = new Etiqueta();
    $etiquetaNueva->nombre = $request->nombre;

    $etiquetaNueva->save();

    $tareas = $request->tareas;
    //Conseguimos el id de esta etiqueta que acabamos de crear
    $etiquetaId = $etiquetaNueva->id;
    $etiquetaNueva->tareas()->attach($tareas, ['etiquetas_id' => $etiquetaId]);

    return new EtiquetaResource($etiquetaNueva);
}
```

En postman, ponemos los datos de la nueva etiqueta en Body > raw:

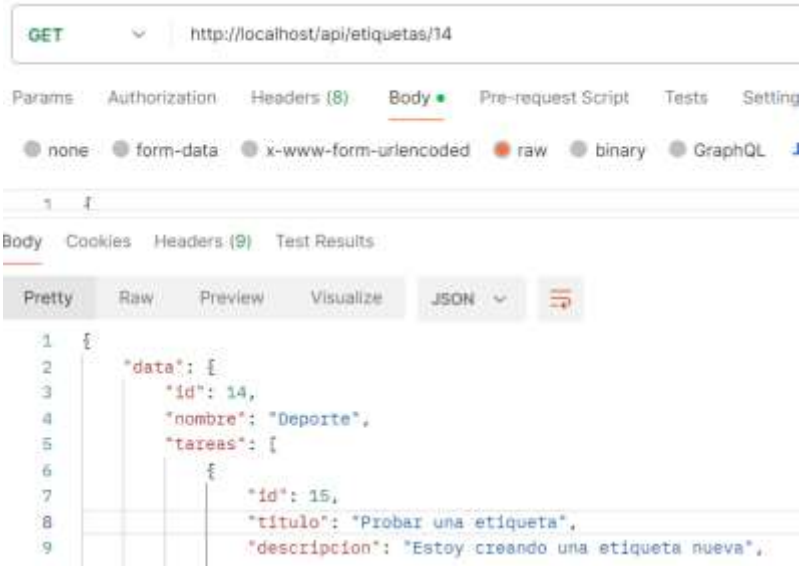
Resultado:

- Función show:

La función show actúa como la función index, pero sólo muestra los datos de una etiqueta concreta. Para llamarla, es necesario realizar un get en /etiquetas/<idEtiqueta>

```php
public function show($idEtiqueta)
{
    $etiqueta = Etiqueta::find($idEtiqueta);
    return new EtiquetaResource($etiqueta);
}
```

Este es el resultado de consultar el endpoint:

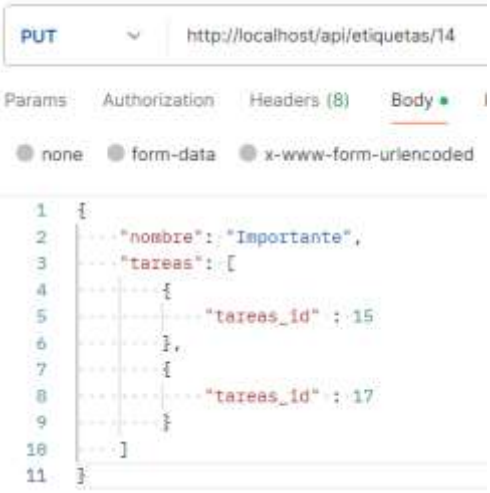


- Función update:

Como su nombre indica, la función update actualiza los datos de una etiqueta en concreto. Esta función se llama al realizar una consulta put especificando el id de la etiqueta: /etiquetas/<idEtiqueta>

```php
public function update(EtiquetaRequest $request, $idEtiqueta)
{
    $etiquetaModif = Etiqueta::find($idEtiqueta);
    $etiquetaModif->nombre = $request->nombre;

    $etiquetaModif->tareas()->detach();
    $etiquetaModif->tareas()->attach($request->tareas);

    $etiquetaModif->save();

    return new EtiquetaResource($etiquetaModif);
}
```

Esta es la consulta que debemos realizar en el postman:



Este es el resultado, cambiando el título por uno nuevo y añadiendo una nueva etiqueta:

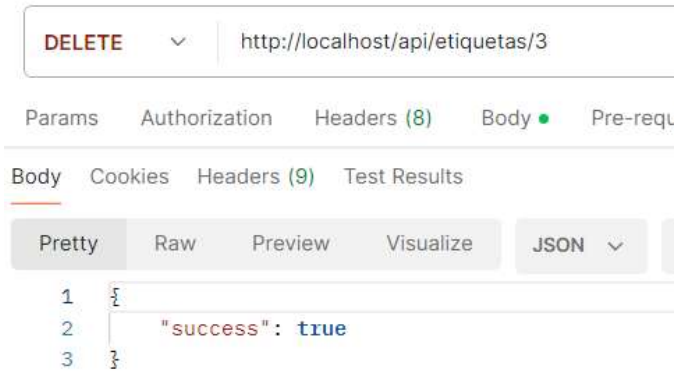

```
{
    "data": {
        "id": 14,
        "nombre": "Importante",
        "tareas": [
            {
                "id": 15,
                "titulo": "Probar una etiqueta",
                "descripcion": "Estoy creando una etiqueta nueva",
                "pivot": {
                    "etiquetas_id": 14,
                    "tareas_id": 15
                }
            },
            {
                "id": 17,
                "titulo": "Limpiar",
                "descripcion": "Fregar el salón y recoger la cocina.",
                "pivot": {
                    "etiquetas_id": 14,
                    "tareas_id": 17
                }
            }
        ]
```


- Función destroy:

La función destroy borra una etiqueta concreta. Se llama a esta función con el endpoint delete /etiquetas/<idEtiqueta>

```php
public function destroy($idEtiqueta)
{
    $etiquetaBorrar = Etiqueta::find($idEtiqueta);
    $etiquetaBorrar->delete();
    return response()->json(['success' => true], 200);
}
```

Resultado:


DELETE
http://localhost/api/etiquetas/3
Params
Authorization
Headers (8)
Body
Pre-requ
Body
Cookies
Headers (9)
Test Results
Pretty
Raw
Preview
Visualize
JSON
1 {
2 "success": true
3 }

3. Usuarios

A continuación, haremos los cambios necesarios en el proyecto para poder realizar un registro de nuevo usuario, login y logout. Además, utilizaremos laravel/sanctum para realizar autentificaciones que sólo nos permitan realizar la llamada a ciertos endpoints con el token de usuario correcto.

Es importante mencionar que a la hora de crear el proyecto de laravel, la tabla users ha sido creada. Con ella, se ha creado un modelo User.

3.1. Preparar laravel/sanctum

Laravel/sanctum se trata de la herramienta que utilizaremos para realizar la autorización a la hora de cerrar el usuario y de realizar las consultas a endpoint de tareas y etiquetas.

Para descargarlo, utilizaremos el comando “sail composer require laravel/sanctum”

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea$
sail composer require laravel/sanctum
el/sanctum
Running composer update laravel/sanctum
Loading composer repositories with package information
Updating dependencies
Nothing to modify in lock file
Writing lock file
Installing dependencies from lock file (including require-dev)
Nothing to install, update or remove
Generating optimized autoload files
```

podemos seguir con la configuración utilizando el comando "sail artisan vendor:publish --provider="Laravel\Sanctum\SanctumServiceProvider"". Este comando publicará los archivos de configuración y migraciones necesarios para integrar Sanctum en nuestra aplicación.

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea$
sail artisan vendor:publish --provider="Laravel\Sanctum\SanctumServiceProvider"

  INFO  Publishing assets.

  Copying directory [vendor/laravel/sanctum/database/migrations] to [database/migrations]
 ..................................................... DONE
  File [config/sanctum.php] already exists ..........................................
 ..................................................... SKIPPED
```

3.2. Crear el modelo de autorización

Una vez hecha la configuración de laravel/sanctum, ya que la tabla de users y el modelo User ya existen, queda crear el controlador de autorización. He decidido no crear resource ni request específicas de User debido a que esta tabla no

funciona de la misma manera que las demás, y cada función de su controlador devuelve un resultado completamente diferente al resto.

Para crear el modelo, utilizamos el comando “sail artisan make:controller AuthController”.

Como he mencionado antes, para que las funciones de AuthController sean llamadas, debemos llamarlas desde api.php. En este caso, las rutas serían las siguientes:

```php
Route::post('register', [AuthController::class, 'register']);
Route::post('login', [AuthController::class, 'login']);

Route::middleware('auth:sanctum')->group(function(){
Route::get('logout', [AuthController::class, 'logout']);
});
```

Podemos ver que se requiere el token de acceso para el logout pero no para el registro ni para el login, ya que no tendremos acceso al token hasta que creemos un usuario o accedamos a uno ya creado.

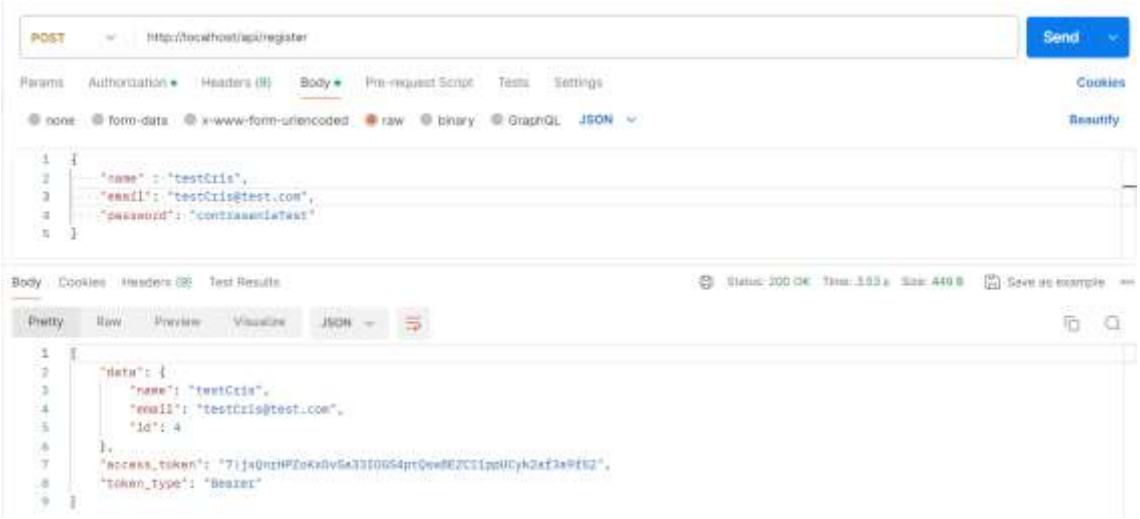
En el archivo del controlador, crearemos estas funciones:

- Función register:

Esta función, que sirve para registrar un nuevo usuario, responde al endpoint post /register. En él, creamos un nuevo usuario, un token para este, y devolvemos los datos del usuario, su token de acceso y el tipo de token.

```php
public function register(Request $request){
    $user = User::create([
        'name'=>$request->name,
        'email'=>$request->email,
        'password'=>Hash::make($request->password)
    ]);
    $token = $user->createToken('auth_token')->plainTextToken;

    return response()->json(['data'=>$user, 'access_token'=>$token, 'token_type'=>'Bearer']);
}
```

El resultado de utilizar el endpoint post /register es el siguiente:

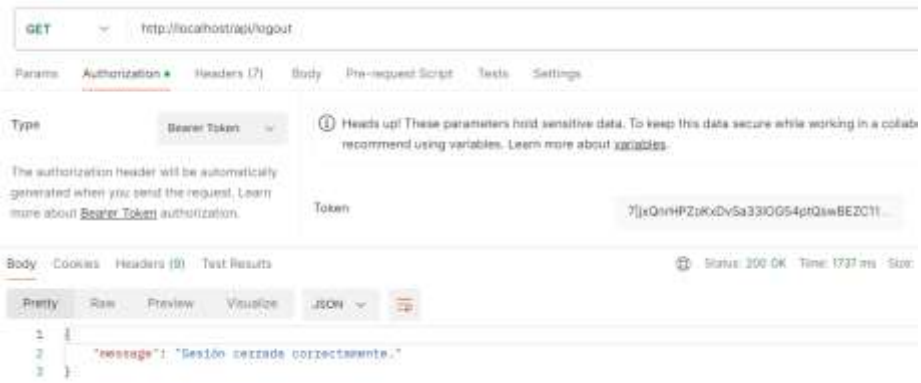


- Función logout:

Esta función, que sirve para cerrar sesión del usuario, responde al endpoint get /logout. Borra el token de autorización del usuario, haciendo que tenga que volver a iniciar sesión para acceder a los datos. Devuelve, además, un mensaje de éxito.

```
1 reference | 0 overrides | Codiumate: Options | Test this method
public function logout(){
    auth()->user()->tokens()->delete();
    return ['message' => 'Sesión cerrada correctamente.'];
}
```

El resultado de utilizar el endpoint get /logout es el siguiente:

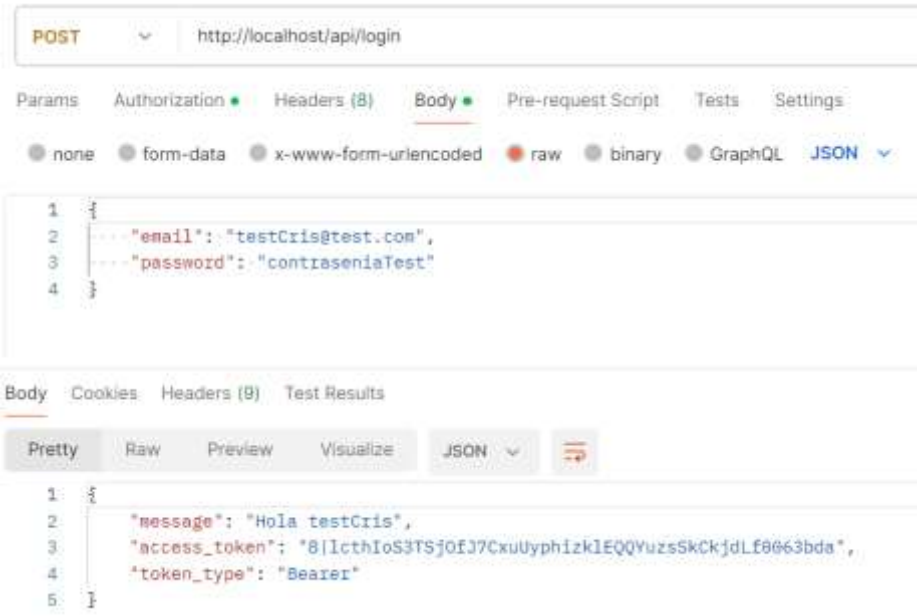


- Función login:

Esta función, que sirve para acceder a un usuario creado anteriormente, responde al endpoint post /login. En él, se busca un usuario con el email proporcionado en la consulta. Si no existe, falla, pero si existe, comprueba la contraseña. Si es correcta, devuelve “Hola, <nombre del usuario>” seguido del token de acceso y del tipo de token. Si no, da un mensaje de credenciales incorrectas.

```php
public function login(Request $request){
    $user = User::where('email', $request->email)->firstOrFail();

    if(!Hash::check($request->password, $user->password)){
        return response()->json(['message' => 'Credenciales incorrectas'], 401);
    }

    $token = $user->createToken('auth_token')->plainTextToken;

    return response()->json(['message'=>'Hola ' . $user->name,
                             'access_token' => $token,
                             'token_type' => 'Bearer']);
}
```

El resultado de utilizar el endpoint post /login es el siguiente:



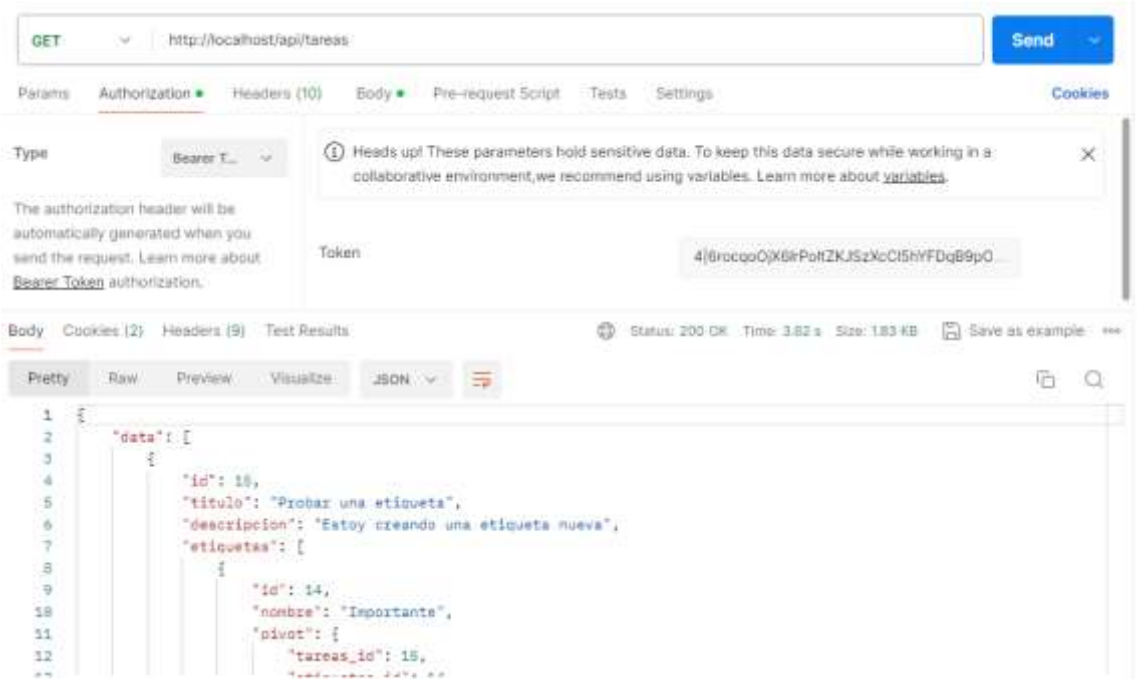
4. Restringir acceso a tareas y etiquetas

Una vez hemos creado todo lo necesario para tener una conexión a la tabla users y recibir un token de acceso, podemos restringir el acceso a las tablas tareas y usuarios para que sólo los usuarios que estén conectados puedan acceder a ellas.

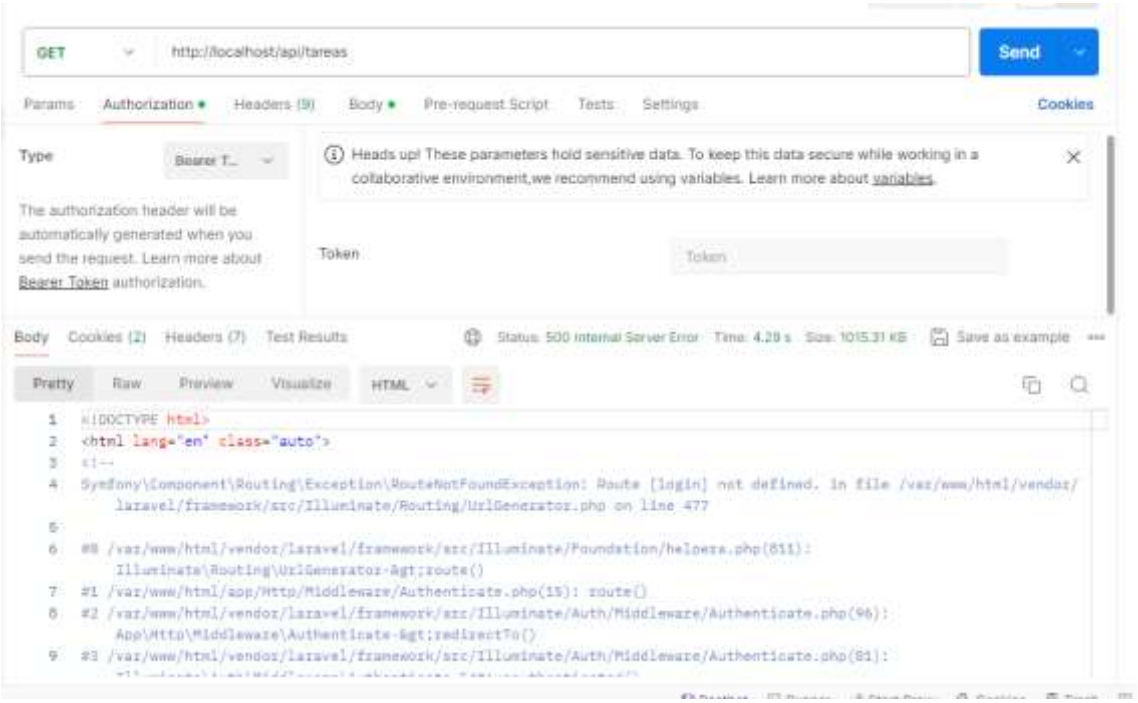
Para esto, debemos añadir las rutas de los endpoints de tareas y etiquetas en el archivo api.php a Route::middleware('auth:sanctum'), como podemos ver a continuación:

```php
Route::post('register', [AuthController::class, 'register']);
Route::post('login', [AuthController::class, 'login']);

Route::middleware('auth:sanctum')->group(function(){
    Route::get('logout', [AuthController::class, 'logout']);
    Route::resource('/tareas',TareaController::class);
    Route::resource('/etiquetas',EtiquetaController::class);
});
```

Una vez hecho esto, necesitaremos utilizar un token de acceso válido en la autorización para acceder a las tareas y etiquetas, y poder realizar acciones CRUD sobre las tablas (Creación, lectura, actualización y borrado).



Podemos ver que si realizamos la misma consulta sin utilizar ningún token, nos da un error.


GET
http://localhost/api/tareas
Send
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
Cookies
Type
Bearer T...
Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment,we recommend using variables. Learn more about variables.
The authorization header will be automatically generated when you send the request. Learn more about Bearer Token authorization.
Token
Token
Body
Cookies (2)
Headers (7)
Test Results
Status: 500 Internal Server Error
Time: 4.29 s
Size: 1015.31 KB
Save as example
Pretty
Raw
Preview
Visualize
HTML
1 <!DOCTYPE html>
2 <html lang="en" class="auto">
3 <!--
4 Symfony\Component\Routing\Exception\RouteNotFoundException: Route [login] not defined. in file /var/www/html/vendor/laravel/framework/src/Illuminate/Routing/UrlGenerator.php on line 477
5
6 #0 /var/www/html/vendor/laravel/framework/src/Illuminate/Foundation/helpers.php(811): Illuminate\Routing\UrlGenerator->route()
7 #1 /var/www/html/app/Http/Middleware/Authenticate.php(15): route()
8 #2 /var/www/html/vendor/laravel/framework/src/Illuminate/Auth/Middleware/Authenticate.php(96): App\Http\Middleware\Authenticate->redirectTo()
9 #3 /var/www/html/vendor/laravel/framework/src/Illuminate/Auth/Middleware/Authenticate.php(81):

5. Realización de tests

Los tests son una herramienta de Laravel que nos permiten realizar pruebas sobre los endpoints para asegurar la integridad y funcionalidad de nuestra aplicación.

5.1. Crear Factories

Los factories son clases de laravel que nos permiten crear una nueva instancia de un objeto concreto que sólo existirá dentro del test y que nos permitirá realizar tests con datos creados por él. Esto permitirá que podamos realizar los tests en diferentes ocasiones con diferentes datos generados automáticamente.

5.1.1. Factory de User

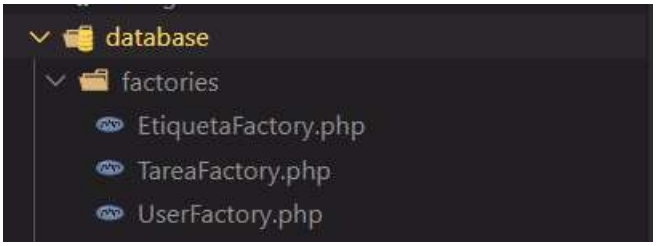
El archivo UserFactory ya viene creado por defecto en laravel. Lo utilizaremos en los tests para crear un token que permita acceder a los datos de las tablas tareas y etiquetas.

5.1.2. Factory de Tarea

Creamos el archivo TareaFactory con el comando “sail artisan make:factory TareaFactory”.

```
cristina@Cristinalaptop:/mnt/c/Users/rhiap/Desktop/2DAW/DWEServidor/docker/www/apiTarea$ sail artisan make:factory TareaFactory

  INFO  Factory [database/factories/TareaFactory.php] created successfully.
```

Una vez creado el archivo, podemos acceder a él en database > factories



En la función definition, utilizamos la función faker para crear los tipos de textos que queramos. En este caso, se creará una palabra para título y una frase para descripción.

```php
    public function definition(): array
    {
        return [
            'titulo' => $this->faker->word,
            'descripcion' => $this->faker->sentence,
        ];
    }
}
```

5.1.3. Factory de Etiqueta

Creamos el archivo EtiquetaFactory con el comando “sail artisan make:factory EtiquetaFactory”.

Esta vez, en definition, crearemos una palabra para la columna nombre.

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

5.2. Creación de tests

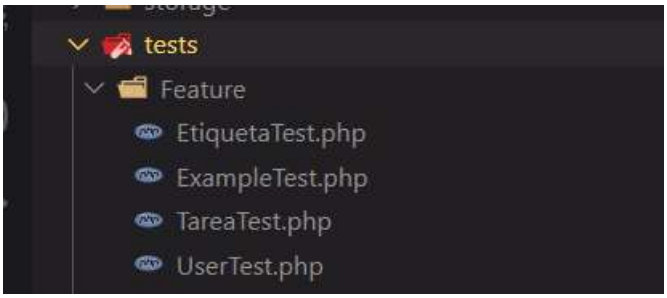
Una vez creados los Factories necesarios, podemos proceder a crear los tests para tareas, etiquetas y users.

5.2.1. Tests de Tarea

Ejecutamos el comando “sail artisan make:test TareaTest” para crear la clase que realizará los tests.

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/api
Tarea$ sail artisan make:test TareaTest

  INFO  Test [tests/Feature/TareaTest.php] created successfully.
```

Este archivo se creará en la carpeta tests > Feature



En este archivo, realizaremos una función test para cada uno de los endpoints a los que responde la tabla tareas.

Antes de comenzar a explicar los tests por separado, es importante comentar el principio de estos tests.

```php
//Creo un usuario con su token para usarlo en la autorización
$user = User::factory()->create();
$token = $user->createToken('test_token');

$header = [
    'Authorization' => 'Bearer ' . $token->plainTextToken
];
```

En esta parte del código, creamos un usuario con UserFactory. Una vez hecho, generamos un token, que utilizaremos en la autorización y que será llamado por la respuesta, la variable $response por medio del código “withHeaders($header)”.

Por otro lado, utilizaremos “use RefreshDatabase” para que los tests no interfieran unos con otros y para que la base de datos de los tests sea consistente.

```php
class TareaTest extends TestCase
{

    use RefreshDatabase;
    /**
```

Para que una función sea tratada como un test, debe tener la palabra “test” al principio de su nombre. Ejemplo: function testMostrarTareas();

- Test mostrar todas las tareas

En el test, podemos ver que se crea un token. Usándolo en la respuesta, llamamos al endpoint get /api/tareas. Si se ha devuelto la información adecuadamente, el estado será 200 y el test se pasará.

```php
public function testMostrarTodasTareas()
{
    //Creo un usuario con su token para usarlo en la autorización
    $user = User::factory()->create();
    $token = $user->createToken('test_token');

    $header = [
        'Authorization' => 'Bearer ' . $token->plainTextToken
    ];

    $response = $this->withHeaders($header)->get('/api/tareas');

    $response->assertStatus(200)
        ->assertJsonStructure([
            'data' => [
                '*' => [
                    'id',
                    'titulo',
                    'descripcion',
                    'etiquetas'
                ],
            ],
        ]);
}
```

- Test mostrar tarea

Este test funciona de forma muy similar al anterior, pero en él utilizamos TareaFactory para generar una tarea de la que poder sacar el id. Este id lo utilizamos más adelante para poder recibir una respuesta al endpoint get utilizando /api/tareas/<idTarea>

- Test crear tarea

En este test, creamos una tarea nueva a la que damos un título y descripción. Mandamos esta tarea en el post /api/tareas. Si se ha creado adecuadamente, el estatus será 201.

- Test modificar tarea

En este test, comprobamos que se pueda actualizar una tarea. Para ello, respondemos al endpoint put /api/tareas/<idTarea> mandando los datos de la

nueva tarea. La tarea a modificar y su id los obtenemos de la misma manera que en el test de mostrar tarea, y creamos una tarea con los datos que queremos modificar.

```php
public function testActualizarTarea(){
    $user = User::factory()->create();
    $token = $user->createToken('test_token');

    $header = [
        'Authorization' => 'Bearer ' . $token->plainTextToken
    ];

    $tareaOriginal = Tarea::factory()->create();

    $tareaModif = [
        'titulo' => 'Nuevo titulo',
        'descripcion' => 'Nueva descripción',
    ];

    $response = $this->withHeaders($header)->put('/api/tareas/' . $tareaOriginal->id, $tareaModif);

    $response->assertStatus(200)
    ->assertJsonStructure([
        'data' => [
            'id',
            'titulo',
            'descripcion',
            'etiquetas'
        ],
    ]);
}
```

- Test borrar tarea

Por último, realizamos un test de borrar tarea. Para esto, creamos una tarea utilizando TareaFactory y luego la borramos, pasando su id como parámetro al llamar al endpoint.

```php
public function testBorrarTarea(){
    $user = User::factory()->create();
    $token = $user->createToken('test_token');

    $header = [
        'Authorization' => 'Bearer ' . $token->plainTextToken
    ];

    $tareaBorrar = Tarea::factory()->create();

    $response = $this->withHeaders($header)->delete('/api/tareas/' . $tareaBorrar->id);

    $response->assertStatus(200)
    ->assertJsonStructure([
        'success'
    ]);
}
```

5.2.2. Creación de tests de Etiqueta

Creamos el archivo de tests con “sail artisan make:test EtiquetaTest”.

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/api
Tarea$ sail artisan make:test EtiquetaTest

  INFO  Test [tests/Feature/EtiquetaTest.php] created successfully.
```

Al igual que en TareaTest, los tests comienzan con la creación de un usuario utilizando UsuarioFactory. También añadiremos 'use RefreshDatabase'.

- Test mostrar todas las etiquetas

Llamamos al endpoint get /api/etiquetas. Si se ha devuelto la información adecuadamente, el estado será 200 y el test se pasará.

```php
public function testMostrarTodasEtiquetas()
{
    //Creo un usuario con su token para usarlo en la autorización
    $user = User::factory()->create();
    $token = $user->createToken('test_token');

    $header = [
        'Authorization' => 'Bearer ' . $token->plainTextToken
    ];

    $response = $this->withHeaders($header)->get('/api/etiquetas');

    $response->assertStatus(200)
        ->assertJsonStructure([
            'data' => [
                '*' => [
                    'id',
                    'nombre',
                    'tareas'
                ],
            ],
        ]);
}
```

- Test mostrar etiqueta

Creamos una etiqueta con EtiquetaFactory para llamar al endpoint get /api/etiquetas/<idEtiqueta>

```php
public function testMostrarEtiqueta()
{
    // Creo un usuario con su token para usarlo en la autorización
    $user = User::factory()->create();
    $token = $user->createToken('test_token');

    $header = [
        'Authorization' => 'Bearer ' . $token->plainTextToken
    ];

    // Creo la etiqueta que mostraría usando el método factory en el modelo Etiqueta
    $etiqueta = Etiqueta::factory()->create();

    $response = $this->withHeaders($header)->get('/api/etiquetas/' . $etiqueta->id);

    $response->assertStatus(200)
        ->assertJsonStructure([
            'data' => [
                'id',
                'nombre',
                'tareas'
            ],
        ]);
}
```

- Test crear etiqueta

En este test, creamos una etiqueta nueva a la que damos un nombre. Mandamos esta etiqueta en el post /api/etiquetas. Si se ha creado adecuadamente, el estatus será 201.

```php
public function testCrearNuevaEtiqueta()
{
    $user = User::factory()->create();
    $token = $user->createToken('test_token');

    $header = [
        'Authorization' => 'Bearer ' . $token->plainTextToken
    ];

    $nuevaEtiqueta = [
        'nombre' => 'Nueva Etiqueta'
    ];

    $response = $this->withHeaders($header)->post('/api/etiquetas', $nuevaEtiqueta);

    $response->assertStatus(201) //201 -> Código para fila creada
        ->assertJsonStructure([
            'data' => [
                'id',
                'nombre',
                'tareas'
            ],
        ]);

}
```

- Test modificar etiqueta

Creamos una etiqueta con EtiquetaFactory para utilizar su id en el endpoint put /api/etiquetas/<idEtiqueta>, y una etiqueta con el nuevo nombre.

```php
public function testActualizarEtiqueta(){
    $user = User::factory()->create();
    $token = $user->createToken('test_token');

    $header = [
        'Authorization' => 'Bearer ' . $token->plainTextToken
    ];

    $etiquetaOriginal = Etiqueta::factory()->create();

    $etiquetaModif = [
        'nombre' => 'Nuevo Nombre'
    ];

    $response = $this->withHeaders($header)->put('/api/etiquetas/' . $etiquetaOriginal->id, $etiquetaModif);

    $response->assertStatus(200)
        ->assertJsonStructure([
            'data' => [
                'id',
                'nombre',
                'tareas'
            ],
        ]);
}
```

- Test borrar etiqueta

Creamos una etiqueta con EtiquetaFactory y tomamos su id en el endpoint delete /api/etiquetas/<idEtiqueta> para borrarla.

```php
public function testBorrarEtiqueta(){
    $user = User::factory()->create();
    $token = $user->createToken('test_token');

    $header = [
        'Authorization' => 'Bearer ' . $token->plainTextToken
    ];

    $etiquetaBorrar = Etiqueta::factory()->create();

    $response = $this->withHeaders($header)->delete('/api/etiquetas/' . $etiquetaBorrar->id);

    $response->assertStatus(200)
    ->assertJsonStructure([
        'success'
    ]);
}
```

5.2.2. Creación de tests de User

Creamos el fichero UserTest.php utilizando el comando “sail artisan make:test UserTest”.

```
cristina@Cristinalaptop:/mnt/c/users/rhiap/desktop/2DAW/DWEServidor/docker/www/apiTarea$ sail artisan make:test UserTest

  INFO  Test [tests/Feature/UserTest.php] created successfully.
```

Al igual que en las clases tipo test anteriores, utilizaremos RefreshDatabase para evitar cualquier problema con la base de datos de tests.

- Test registrar usuario

En este test, creamos un nuevo usuario con nombre, email y contraseña. Lo mandamos en la respuesta al endpoint post /api/register.

```php
public function testRegistrarUsuario(){
    $nuevoUsuario = [
        'name' => 'test',
        'email' => 'test@mail.com',
        'password' => 'contrasenia'
    ];

    $response = $this->post('/api/register', $nuevoUsuario);

    $response->assertStatus(200)
    ->assertJsonStructure([
        'data' => [
            'id',
            'name',
            'email'
        ],
    ]);
}
```

- Test login

En este test, creamos un usuario con contraseña ‘password’. Esto es necesario ya que necesitamos conocer la contraseña para añadirla en las credenciales que utilizaremos para acceder.

```php
public function testLogin(){
    $usuarioLogin = User::factory()->create(['password' => 'password']);

    $credenciales = [
        'email' => $usuarioLogin->email,
        'password' => 'password'
    ];

    $response = $this->post('/api/login', $credenciales);

    $response->assertStatus(200)
    ->assertJsonStructure([
        'message',
        'access_token',
        'token_type',
    ])
    ->assertJson([
        'message' => 'Hola ' . $usuarioLogin->name,
        'token_type' => 'Bearer',
    ]);

}
```

- Test logout

En este test, crearemos un usuario y obtendremos su token para poder proceder. Una vez lo tenemos, solo queda utilizarlo como forma de autorización al llamar al header. El test funciona al llamar al endpoint get /api/logout.

```php
public function testLogout(){
    $usuarioConectado = User::factory()->create();
    $token = $usuarioConectado->createToken('test_token');

    $header = [
        'Authorization' => 'Bearer ' . $token->plainTextToken
    ];

    $response = $this->withHeaders($header)->get('/api/logout');

    $response->assertStatus(200)
    ->assertJson([
        'message' => 'Sesión cerrada correctamente.'
    ]);
}
```

5.3. Ejecutar los tests

Para ejecutar los tests, utilizamos el comando “sail artisan test”.

Si todo ha salido bien, este será el resultado: