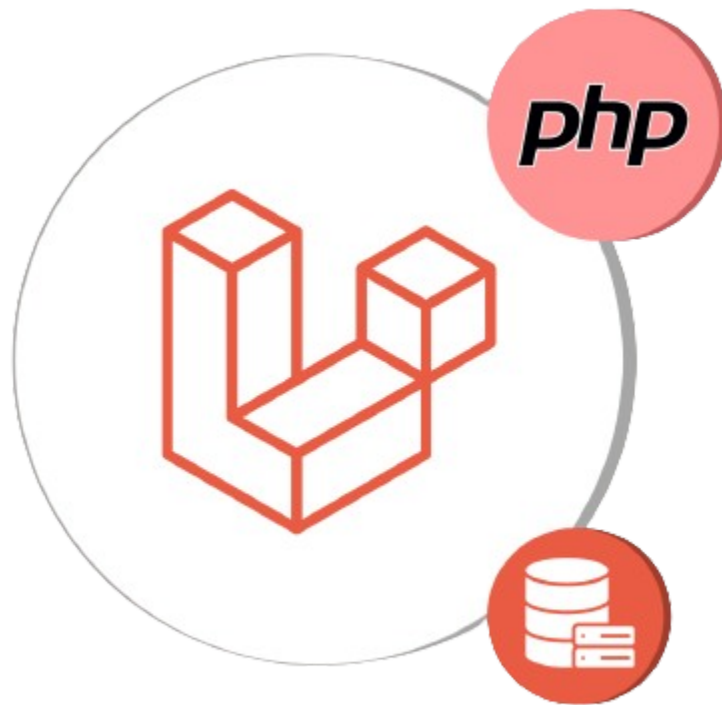


# Desarrollo web en entorno servidor: Desarrollando un proyecto en Laravel



---

**Realizado por : Irene Martín Barea**

**Asignatura : Desarrollo web en entorno servidor (DWES)**

**Profesor : Alicia Vega Moreno**

**Fecha : 03/03/24**

## Índice

|   |           |
|---|-----------|
| <b>1. Creación del proyecto en la terminal de Ubuntu.....</b> | <b>3</b>  |
| 1.1 Comando composer.....                                     | 3         |
| 1.2 Comandos Laravel sail.....                                | 6         |
| 1.3 Carpetas importantes en la estructura del proyecto.....   | 6         |
| <b>2. Creación de la base de datos.....</b>                   | <b>6</b>  |
| 2.1 Migraciones.....  | 6         |
| 2.2 Seeders.....  | 10        |
| <b>3. Creación de los archivos de http.....</b>               | <b>13</b> |
| 3.1 Modelos.....  | 13        |
| 3.2 Resources.....  | 15        |
| 3.3 Requests.....   | 16        |
| 3.4 Controladores.....  | 18        |
| <b>4. Creación del login, register y logout.....</b>          | <b>20</b> |
| 4.1 Instalación del paquete Laravel Sanctum.....              | 20        |
| 4.2 Creación de AuthController.....                           | 21        |
| 4.3 Rutas protegidas por Sanctum.....                         | 23        |
| <b>5. Creación de los tests.....</b>                          | <b>24</b> |
| <b>6. Comprobación en postman.....</b>                        | <b>28</b> |
| 6.1 Comprobación de registro, login y logout.....             | 28        |
| 6.2 Comprobación de tareas.....                               | 31        |
| 6.2 Comprobación de etiquetas.....                            | 35        |

# 1. Creación del proyecto en la terminal de Ubuntu

## 1.1 Comando composer

Una vez hemos creado nuestro directorio de trabajo para la utilización de Laravel, ya podemos pasar a la creación de nuestro primer proyecto. Para ello, utilizaremos la instrucción **Composer** en el siguiente comando:

❖ **composer create-project --prefer-dist laravel/laravel <nombre\_proyecto>**

\* La opción **--prefer-dist** se usa para instalar la última versión del framework. Sin embargo, podemos elegir la versión que deseemos, además de la subversión más actualizada de esta con (\*).

❖ **Ejemplo → composer create-project laravel/laravel "5.8.\*"**

Hay que recordar que tenemos que estar en la ubicación de nuestra área de trabajo, que en nuestro caso sería la ruta en la que tenemos almacenada la carpeta **Docker\_compose/www**.

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www$ composer create-project --prefer-dist laravel/laravel APItareas
```

```
Deprecation Notice: Using ${var} in strings is deprecated, use {%var} instead in /usr/share
/php/Symfony/Component/Console/Command/DumpCompletionCommand.php:48
Deprecation Notice: Using ${var} in strings is deprecated, use {%var} instead in /usr/share
/php/Symfony/Component/Console/Command/DumpCompletionCommand.php:56
Creating a "laravel/laravel" project at "./APItareas"
Deprecation Notice: Using ${var} in strings is deprecated, use {%var} instead in /usr/share
/php/Composer/Autoload/AutoloadGenerator.php:879
Deprecation Notice: Using ${var} in strings is deprecated, use {%var} instead in /usr/share
/php/Composer/Autoload/AutoloadGenerator.php:884
Installing laravel/laravel (v10.3.2)
- Installing laravel/laravel (v10.3.2): Extracting archive
Created project in /mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Doc
ker_compose/www/APItareas
```

## 1.2 Comandos laravel sail

Ahora ya tenemos nuestro proyecto, por lo tanto tenemos que acceder a la nueva carpeta creada con el nombre del proyecto y descargar **laravel sail** utilizando el comando **artisan**. Teniendo estas dependencias activadas podremos realizar los próximos pasos para crear la estructura del proyecto\*.

## Instalación de un servidor ubuntu en AWS

El comando de instalación es el siguiente:

❖ **php artisan sail:install**

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/APItareas$ php artisan sail:install

Which services would you like to install? _____
mysql
```

Tras la instalación:

```
INFO Sail scaffolding installed successfully. You may run your Docker containers using Sail's
"up" command.

→ ./vendor/bin/sail up

WARN A database service was installed. Run "artisan migrate" to prepare your database:

→ ./vendor/bin/sail artisan migrate
```

---

\* En el caso de que haya ocurrido algún error y no se hayan descargado las dependencias, podemos utilizar el siguiente comando:

❖ **composer require laravel/sail --dev**

---

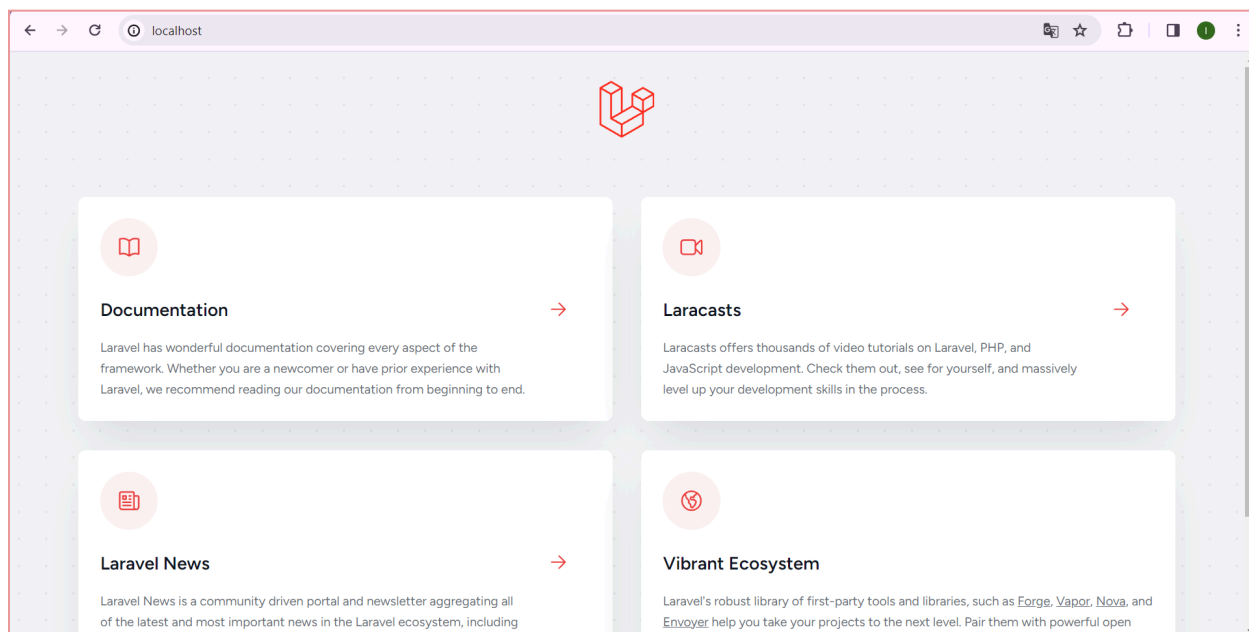
Una vez realizada la descarga y asegurándonos de tener el Docker abierto, ya podemos ejecutar la herramienta de la siguiente forma:

❖ **./vendor/bin/sail up -d → (-d para que se ejecute en segundo plano)**

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/APItareas$ ./vendor/bin/sail up -d
[+] Running 4/4
✓ Network apitareas_sail          Created      0.1s
✓ Volume "apitareas_sail-mysql"   Created      0.0s
✓ Container apitareas-mysql-1     Started      0.1s
✓ Container apitareas-laravel.test-1 Started      0.1s
```

Tras hacer esto, podemos comprobar que funciona correctamente buscando **Localhost** desde el browser de nuestro navegador, que nos muestra una web por defecto que nos genera el instalador de laravel. Si esto no es así, es posible que se dé el caso de que tengamos por ejemplo el servidor de apache ocupando el puerto, en este caso debemos de parar dicho servidor con:

❖ **sudo systemctl stop apache2**



Dado que el comando **sail** lo utilizaremos siempre que queramos realizar una acción con Laravel es recomendable aplicarle un alias a **./vendor/bin/sail** para acortarlo únicamente a **sail**. Para ello, escribimos lo siguiente:

❖ **alias sail="./vendor/bin/sail"**

\* Si queremos saber la lista de los comandos que se pueden realizar con laravel sail llamamos directamente al comando **sail**. A continuación se pueden ver algunos de los que aparecen:

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker/_compose/www/APItareas$ sail
Laravel Sail

Usage:
  sail COMMAND [options] [arguments]

Unknown commands are passed to the docker-compose binary.

docker-compose Commands:
  sail up           Start the application
  sail up -d        Start the application in the background
  sail stop         Stop the application
  sail restart      Restart the application
  sail ps           Display the status of all containers

Artisan Commands:
  sail artisan ...   Run an Artisan command
  sail artisan queue:work
```

## 1.3 Carpetas importantes en la estructura del proyecto

Al crear nuestro proyecto de Laravel con el comando composer se nos añade automáticamente una estructura básica con una gran variedad de archivos y directorios. Algunos de los más importantes que deberíamos reconocer son los siguientes:

- ❖ **app/http** → contiene las carpetas **Controllers**, **Middlewares**, **Requests** y **Resources**
- ❖ **app/Models** → contiene los archivos de los modelos para aplicar la arquitectura Modelo/Vista/Controlador.
- ❖ **app/Providers**
- ❖ **app/bootstrap**
- ❖ **app/config**
- ❖ **app/database** → contiene las carpeta **factories**, **migrations** y **seeders**
- ❖ **app/routes**
- ❖ **app/tests**
- ❖ **Archivo .env** → Es el archivo en el que se encuentran las variables de entorno, como por ejemplo la conexión a la base de datos.

## 2. Creación de la base de datos

### 2.1 Migraciones

Las migraciones se utilizan para crear y administrar la estructura de nuestra base de datos, de forma que podamos definir cambios haciendo que sean fáciles de seguir y controlar, por ejemplo permitiendo el control de versiones.

Con las migraciones podemos crear un objeto que defina las tablas de la base de datos y para ello utilizamos el siguiente comando:

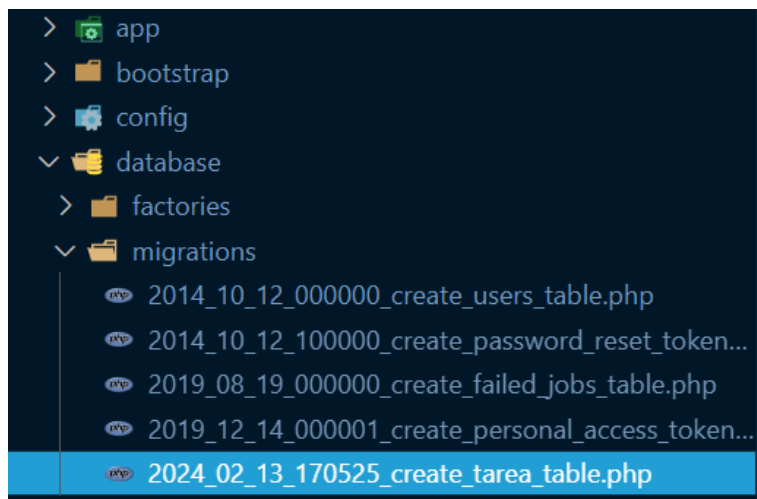
- ❖ **sail artisan make:migration create\_<nombre\_tabla>\_table**

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/APItareas$ sail artisan make:migration create_tarea_table

INFO Migration [database/migrations/2024_02_13_170525_create_tarea_table.php] created successfully.
```

Estas tablas se almacenarán en uno de los directorios mencionados anteriormente:  
**app/database/migrations.**

## Instalación de un servidor ubuntu en AWS



Si abrimos el archivo creado nos aparecerá automáticamente el siguiente código:

```
<?php

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

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

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

En la función **up** podemos establecer la estructura de la tabla, creando la clave primaria (al crear la tabla se crea por defecto un id) y las columnas correspondientes que se crearán al hacer la migración. Además también aparece por defecto un método **timestamps** que sirve para crear

dos columnas, **created\_at** y **updated\_at** que serán gestionados por Laravel de forma automática cada vez que se actualice o se inserte un nuevo registro.

Por otro lado la función **down** sirve para eliminar la tabla en caso de que exista, lo cual se ejecuta cada vez que se hace un rollback.

En el caso de esta tabla de tareas crearemos los siguientes campos dentro de la función up para definir un título y una descripción de la tarea:

```
public function up(): void
{
    Schema::create('tarea', function (Blueprint $table) {
        $table->id();
        $table->string("titulo", 20);
        $table->string("descripcion", 200)->nullable;
        $table->timestamps(); /*creación de las columnas: created_at y updated_at*/
    });
}
```

Cuando tenemos la tabla configurada, lo siguiente que hay que hacer es lanzar la migración contra la base de datos para crear dicha tabla dentro de ella. Se utiliza el siguiente comando:

### ❖ **sail artisan migrate**

Este comando lanzará todas las tablas que no estén ya creadas en la base de datos y se encuentren definidas en las migraciones, así que en el caso de que queramos revertir este paso una vez realizado podemos utilizar 2 instrucciones:

- ❖ **sail artisan migrate:rollback** → este comando elimina la última migración realizada.
- ❖ **sail artisan migrate:refresh** → este comando elimina todas las migraciones realizadas y las vuelve a ejecutar de nuevo, siendo la mejor opción en el caso de que se haya cometido algún error en alguna de las migraciones.

### Creación de las demás tablas:\*

Una vez tenemos establecido la tabla de tareas como ejemplo, procederemos a ampliar el proyecto añadiéndole una tabla de etiquetas y una tabla para relacionar las tareas con las etiquetas.

\* En las imágenes aparecen las tablas en singular, sin embargo, se ha modificado esto para que sea al contrario, por lo que en realidad las tablas creadas serían: **create\_tareas\_table**, **create\_etiquetas\_table** y **create\_tareas\_etiquetas**. Es por lo anterior que sirve de bastante utilidad el comando **sail artisan migrate:refresh**.



# Instalación de un servidor ubuntu en AWS

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$ sail artisan make:migration create_etiqueta_table
```

**INFO** Migration [database/migrations/2024\_02\_14\_154653\_create\_etiqueta\_table.php] created successfully.

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$ sail artisan make:migration create_tarea_etiqueta
```

**INFO** Migration [database/migrations/2024\_02\_14\_154908\_create\_tarea\_etiqueta.php] created successfully.

## Ejemplo de migración y refresh:

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$ sail artisan migrate
```

**INFO** Preparing database.

Creating migration table ..... 137ms **DONE**

**INFO** Running migrations.

|   |       |             |
|---|-------|-------------|
| 2014_10_12_000000_create_users_table                  | 133ms | <b>DONE</b> |
| 2014_10_12_100000_create_password_reset_tokens_table  | 24ms  | <b>DONE</b> |
| 2019_08_19_000000_create_failed_jobs_table            | 87ms  | <b>DONE</b> |
| 2019_12_14_000001_create_personal_access_tokens_table | 70ms  | <b>DONE</b> |
| 2024_02_13_170525_create_tarea_table                  | 26ms  | <b>DONE</b> |
| 2024_02_14_154653_create_etiqueta_table               | 27ms  | <b>DONE</b> |

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$ sail artisan migrate:refresh
```

**INFO** Rolling back migrations.

|   |      |                     |
|---|------|---------------------|
| 2024_02_19_164229_create_tareas_etiquetas             | 61ms | <b>DONE</b>         |
| 2024_02_19_162758_create_etiquetas_table              | 14ms | <b>DONE</b>         |
| 2024_02_19_162151_create_tareas_table                 | 13ms | <b>DONE</b>         |
| 2024_02_14_154653_create_etiqueta_table               |      | Migration not found |
| 2024_02_13_170525_create_tarea_table                  |      | Migration not found |
| 2019_12_14_000001_create_personal_access_tokens_table | 19ms | <b>DONE</b>         |
| 2019_08_19_000000_create_failed_jobs_table            | 15ms | <b>DONE</b>         |
| 2014_10_12_100000_create_password_reset_tokens_table  | 17ms | <b>DONE</b>         |
| 2014_10_12_000000_create_users_table                  | 14ms | <b>DONE</b>         |

## Definir el código de las tablas:

- ❖ **Etiquetas** → Para la tabla de etiquetas definiremos un campo de “nombre” de la etiqueta.

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

- ❖ **Tareas etiquetas** → En la tabla de relación entre las tareas y las etiquetas debemos definir las claves foráneas para realizar el vínculo entre ellas.

```
public function up(): void
{
    Schema::create('tareas_etiquetas', function (Blueprint $table) {
        $table->id();
        //Dejarlo preparado para hacer el enganche (la barra baja separa la tabla y la columna)
        //por ejemplo: la tabla tareas tiene una columna id que es una clave foránea
        $table->foreignId("tareas_id")->constrained()->onDelete('cascade'); //clave foranea
        $table->foreignId("etiquetas_id")->constrained()->onDelete('cascade'); //clave foranea
        $table->timestamps();
    });
}
```

Utilizamos la función **constrained()** para establecer la relación de clave foránea, relacionando la tabla con la columna, según la barra baja que se usa como separación de una y otra. Por eso es importante que los nombres “**tareas\_id**” y “**etiquetas\_id**” coincidan con los nombres de sus tablas correspondientes:

- ❖ “**tareas\_id**” → La tabla **tareas** tiene una columna **id**, que es la clave foránea.
- ❖ “**etiquetas\_id**” → La tabla **etiquetas** tiene una columna **id**, que es la clave foránea.

Por otro lado la función **onDelete('cascade')** sirve para que en el caso de que se elimine un registro de la tabla referenciada, se propague también al registro de la tabla intermedia de “**tareas\_etiquetas**”.

## 2.2 Seeders

Los seeders se utilizan para realizar la inserción de datos según las tablas creadas por medio de las migraciones. Sabiendo esto, crearemos unos datos iniciales para la tabla de Tareas y la tabla de Etiquetas. Para crear un seeder se utiliza el siguiente comando:

- ❖ **sail artisan make:seeder <nombreDelSeeder>**

### Creación de los seeders de Tareas y Etiquetas:

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$
sail artisan make:seeder TareaSeeder

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

irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$
sail artisan make:seeder EtiquetaSeeder

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

Definición de los datos de los seeders (los archivos se encuentran en la carpeta database):

- ❖ **TareaSeeder** → Definimos los títulos y las descripciones de las tareas iniciales, haciendo referencia a la tabla **"tareass"**.

```
class TareaSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        DB::table('tareass')->insert([
            'titulo'=>'DWES',
            'descripcion' => 'Estudiar php y laravel.'
        ]);
        DB::table('tareass')->insert([
            'titulo'=>'Películas',
            'descripcion' => 'Hacer maratón de películas con los amigos.'
        ]);
        DB::table('tareass')->insert([
            'titulo'=>'Cena',
            'descripcion' => 'Preparar pizza para cenar.'
        ]);
    }
}
```

- ❖ **EtiquetaSeeder** → Definimos los nombres de las etiquetas iniciales, haciendo referencia a la tabla **"etiquetas"**.

```
class EtiquetaSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        DB::table('etiquetas')->insert([
            'nombre'=>'Estudio'
        ]);
        DB::table('etiquetas')->insert([
            'nombre'=>'Ocio'
        ]);
        DB::table('etiquetas')->insert([
            'nombre'=>'Hogar'
        ]);
        DB::table('etiquetas')->insert([
            'nombre'=>'Otros'
        ]);
    }
}
```

Tras haber hecho lo anterior, debemos definir también el **DatabaseSeeder** (viene ya creado), donde llamaremos a los otros seeders para poblar las tablas con datos, ya que este es el archivo predeterminado que Laravel ejecuta.

```
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     */
    public function run(): void
    {
        $this->call([TareaSeeder::class, EtiquetaSeeder::class]);
    }
}
```

Por último, debemos hacer la importación de los datos a las tablas ya creadas por medio del siguiente comando:

❖ **sail artisan db:seed**

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/APItareas$
sail artisan db:seed

INFO Seeding database.

Database\Seeders\TareaSeeder ..... RUNNING
Database\Seeders\TareaSeeder ..... 241 ms DONE

Database\Seeders\EtiquetaSeeder ..... RUNNING
Database\Seeders\EtiquetaSeeder ..... 22 ms DONE
```

Para comprobar que efectivamente los datos han sido insertados en las tablas tenemos que poner en marcha la consola de mysql con **sail mysql**.

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/APItareas$
sail mysql
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 40
Server version: 8.0.32 MySQL Community Server - GPL

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> |
```

En esta consola podemos utilizar los selects de las respectivas tablas para ver los datos almacenados:

```
mysql> SELECT * FROM tareas
-> ;
```

| id | titulo    | descripcion                                | created_at | updated_at |
|----|-----------|--|------------|------------|
| 1  | DWES      | Estudiar php y laravel.                    | NULL       | NULL       |
| 2  | Peliculas | Hacer maraton de peliculas con los amigos. | NULL       | NULL       |
| 3  | Cena      | Preparar pizza para cenar.                 | NULL       | NULL       |

```
3 rows in set (0.01 sec)
```

```
mysql> SELECT * FROM etiquetas;
```

| id | nombre  | created_at | updated_at |
|----|---------|------------|------------|
| 1  | Estudio | NULL       | NULL       |
| 2  | Ocio    | NULL       | NULL       |
| 3  | Hogar   | NULL       | NULL       |
| 4  | Otros   | NULL       | NULL       |

```
4 rows in set (0.00 sec)
```

### 3. Creación de los archivos de http

#### 3.1 Modelos

Los modelos en Laravel son clases PHP que interactúan con la base de datos, permitiendo realizar operaciones de consulta y manipulación de datos de manera orientada a objetos. Para crear el modelo de las tareas y etiquetas utilizamos el siguiente comando:

❖ **sail:artisan make:model <nombre\_Modelo> -cr**

---

\*La opción -c se utiliza para crear también un controlador asociado al modelo, del cual hablaremos más adelante.

---

### Creación de los modelos Tarea y Etiqueta:

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$ sail artisan make:model Tarea -cr
INFO Model [app/Models/Tarea.php] created successfully.
INFO Controller [app/Http/Controllers/TareaController.php] created successfully.
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$ sail artisan make:model Etiqueta -cr
INFO Model [app/Models/Etiqueta.php] created successfully.
INFO Controller [app/Http/Controllers/EtiquetaController.php] created successfully.
```

Si abrimos el archivo del modelo nos aparecerá la siguiente estructura de código proporcionada por laravel:

```
<?php

namespace App\Models;

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

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

La declaración “**use HasFactory**” que viene predefinida en un modelo de Laravel es una referencia al trait llamado **HasFactory**. Este trait proporciona métodos y funcionalidades que facilitan la creación de instancias de modelos utilizando factories (fábricas) en Laravel.

Poniendo como ejemplo el modelo **Tarea**, para ver algunas de las indicaciones que le podemos dar a los modelos, aquí es importante definir la relación de muchos a muchos (**belongsToMany**) entre la clase tarea y la clase etiqueta, utilizando la tabla de relación entre ellas (‘tareas\_etiquetas’) y sus respectivos ids (‘tareas\_id’, ‘etiquetas\_id’).

```
class Tarea extends Model
{
    use HasFactory;

    protected $guarded = [];
    protected $hidden = ["created_at", "updated_at"];

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

Además también podemos especificar con **\$hidden = ["created\_at", "updated\_at"]**; que las columnas "created\_at" y "updated\_at", las cuales se crean de forma automática con el método **timestamps()** mostrado anteriormente, no aparezcan cuando el modelo se convierta a formato json.

### 3.2 Resources

En Laravel, un "resource" (recurso) es una forma de formatear y estructurar la salida de una API. Los resources son especialmente útiles cuando se está construyendo una API RESTful y se necesita controlar cómo se presentan los datos cuando son enviados como respuesta a las solicitudes HTTP.

El comando que se utiliza para crear un resource es el siguiente:

❖ **sail artisan make:resource <nombre\_Resource>**

#### Creación de resources para las tareas y las etiquetas:

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$ sail artisan make:resource TareaResource
INFO Resource [app/Http/Resources/TareaResource.php] created successfully.
```

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$ sail artisan make:resource EtiquetaResource
INFO Resource [app/Http/Resources/EtiquetaResource.php] created successfully.
```

A continuación, poniendo como ejemplo **TareaResource** para darle formato a la salida de las tareas podemos definir lo siguiente:

```
class TareaResource extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        //Es lo que tiene que aparecer al hacer el get
        return [
            'id' => $this->id,
            'Titulo' => 'Titulo: '. $this->titulo,
            'descripcion' => 'Desc: '. $this->descripcion,
            //'etiquetas' => $this->etiquetas
            //Si hay etiquetas muestra los nombres, de lo contrario aparece un mensaje
            'etiquetas' => $this->etiquetas != null && $this->etiquetas->isNotEmpty() ?
                $this->etiquetas->pluck('nombre') : 'No hay etiquetas asociadas'
        ];
    }
}
```

Dentro del código inicial aparece de forma predeterminada la estructura de la clase `TareaResource`, que se extiende de la clase base `JsonResource`, y también declara una función `toArray()`, la cual sirve para definir que los datos del modelo `Tarea` deben transformarse en un array para la respuesta `Json`. A su vez, dentro del `return` podemos definir los atributos que se quieren mostrar y cómo presentarlos cuando se haga una solicitud `GET`.

### 3.3 Requests

En Laravel, las "requests" (solicitudes) se refieren a las instancias de la clase `Illuminate\Http\Request`. Estas representan la información de la solicitud `HTTP` recibida por la API. Laravel utiliza objetos `Request` para proporcionar un acceso fácil y estructurado a los datos de la solicitud, como los parámetros de la URL, los datos del formulario, las cookies, las cabeceras y más.

Una de las funcionalidades que ofrecen las requests y que resulta útil para el proyecto de tareas es la validación de los datos de entrada. Con estas validaciones se pueden definir reglas sobre cómo deben estar definidos los datos, por ejemplo al hacer un `POST` para añadir una nueva tarea, para que sean aceptados.

Para crear una nueva request se utiliza el siguiente comando:

❖ **sail artisan make:request <nombre\_Request>**

#### Creación de requests para las tareas y las etiquetas:

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$ sail artisan make:request TareaRequest
INFO Request [app/Http/Requests/TareaRequest.php] created successfully.
```

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$ sail artisan make:request EtiquetaRequest
INFO Request [app/Http/Requests/EtiquetaRequest.php] created successfully.
```

Si utilizamos como ejemplo **TareaRequest** podemos definir en el código las siguientes reglas en la función **rules()**, que viene predeterminada al crear el archivo:

```
public function rules(): array
{
    return [
        'titulo' => 'required|max:20|min:10',
        'descripcion' => 'nullable|max:200|min:10',
    ];
}
```



En este caso estaríamos especificando que en las tareas tiene que haber obligatoriamente (required) un título de como máximo 20 caracteres y 10 como mínimo, y que la descripción de la tarea puede estar vacía (nullable), pero en el caso de que haya debe tener un máximo de 200 caracteres y un mínimo de 10.

Además también es importante hacer una modificación en la función **authorize()** que también viene predefinida en la request. Esta función booleana devuelve por defecto false pero debemos cambiarlo a **true**, de lo contrario, en el momento de hacer las peticiones dará un error diciendo que no estamos autorizados.

```
class TareaRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     */
    public function authorize(): bool
    {
        return true;
    }
}
```

### 3.4 Controladores

En Laravel, un controlador (controller) es una clase que maneja las solicitudes HTTP y contiene la lógica para procesar esas solicitudes y devolver una respuesta. Pueden manejar solicitudes **GET**, **POST**, **PUT**, **DELETE**, entre otras, y ejecutar la lógica correspondiente para cada tipo de solicitud.

Como se ha mencionado anteriormente, los controladores ya los tenemos creados al utilizar la opción **-c** al final del comando de creación del modelo, sin embargo, si quisiéramos crearlo de forma individual se utilizará el siguiente comando:

❖ **sail artisan make:controller <nombre\_controller>**

En estos controladores veremos una serie de funciones principales para responder a las diferentes solicitudes HTTP. En el caso del proyecto de tareas y etiquetas las fundamentales son las siguientes:

- ❖ **index()** → para mostrar un listado de todos los recursos con **GET**.
- ❖ **store()** → para almacenar los recursos nuevos cuando se utilice **POST**.
- ❖ **show()** → para mostrar un recurso específico con **GET**.
- ❖ **update()** → para actualizar un recurso específico utilizando **PUT**.
- ❖ **destroy()** → para eliminar un recurso específico con **DELETE**.

También es importante revisar que el controlador tiene asociados las requests, las resources y sus modelos correspondientes, además del formato JsonResponse. Este es un ejemplo del controlador de tareas:

```
<?php

namespace App\Http\Controllers;

use App\Models\Tarea;
use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;
use App\Http\Requests\TareaRequest;
use App\Http\Resources\TareaResource;
```

### Función index() de Tareas:

```
public function index():JsonResource
{
    //Obtener todos los registros de tareas
    $tareas = Tarea::all();
    //Convierte las tareas recuperadas en una colección utilizando el formato TareaResource
    return TareaResource::collection($tareas);
}
```

### Función store() de Tareas:

\$request es una instancia de TareaRequest, para que los datos de la solicitud cumplan las reglas.

```
public function store(TareaRequest $request):JsonResource
{
    //Crear una nueva instancia de la clase Tarea
    $tarea=new Tarea();

    //Asignar los valores a los atributos de la tarea desde la solicitud
    $tarea->titulo = $request->titulo;
    $tarea->descripcion = $request->descripcion;

    //Guardar la tarea en la base de datos
    $tarea->save();

    //Asociar la tarea con las etiquetas proporcionadas en la solicitud (si las hay)
    $tarea->etiquetas()->attach($request->etiquetas);

    //Devolver la tarea en una instancia con el formato de TareaResource
    return new TareaResource($tarea);
}
```

### Función show() de Tareas:

```
public function show($id):JsonResource
{
    // Encontrar la tarea en la base de datos por su ID
    $tarea = Tarea::find($id);

    // Devolver la tarea en una instancia con el formato de TareaResource
    return new TareaResource($tarea);
}
```

### Función de update() de Tareas:

```
public function update(TareaRequest $request, $id):JsonResource
{
    // Encontrar la tarea en la base de datos por su ID
    $tarea = Tarea::find($id);

    //Asignar los valores a los atributos de la tarea desde la solicitud
    $tarea->titulo = $request->titulo;
    $tarea->descripcion = $request->descripcion;

    // Desasociar todas las etiquetas existentes de la tarea
    $tarea->etiquetas()->detach();

    // Asociar las nuevas etiquetas proporcionadas en la solicitud
    $tarea->etiquetas()->attach($request->etiquetas);
    // Guardar los cambios en la base de datos
    $tarea->save();

    // Devolver la tarea en una instancia con el formato de TareaResource
    return new TareaResource($tarea);
}
```

### Función destroy() de Tareas:

```
public function destroy($id)
{
    // Encontrar la tarea en la base de datos por su ID
    $tarea = Tarea::find($id);

    // Verificar si la tarea fue encontrada
    if ($tarea){
        // Eliminar la tarea de la base de datos
        $tarea->delete();

        //devolver una respuesta JSON de éxito
        return response()->json(['success' => true], 200);
    }else{
        // Si la tarea no fue encontrada, devolver un mensaje de error en formato JSON
        return response()->json(['mensaje' => 'Tarea no encontrada'], 404);
    }
}
```

Otra de las cosas a tener en cuenta para que funcione la API, es asegurarnos de que la rutas que maneje el controlador apunten hacia este en el archivo api.php, además de verificar que se esté usando la clase con use, de lo contrario no podrá encontrar el controlador.

```
use App\Http\Controllers\TareaController;
use App\Http\Controllers\EtiquetaController;
```

En este caso si la ruta apunta a **localhost/api/tareas**, se ejecutará **TareaController** y si apunta a **localhost/api/etiquetas**, se ejecutará **Etiquetacontroller**.

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

## 4. Creación del login, register y logout

### 4.1 Instalación del paquete Laravel Sanctum

Laravel Sanctum es un paquete que proporciona un sistema de autenticación de API para aplicaciones de una sola página (SPA) y aplicaciones móviles. Está diseñado para trabajar con

aplicaciones que utilizan autenticación basada en tokens y proporciona una forma sencilla de emitir tokens de acceso para autenticar las solicitudes a la API.

Para realizar la instalación se utiliza el siguiente comando:

### ❖ sail composer require laravel/sanctum

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$
sail composer require laravel/sanctum
./composer.json has been updated
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
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi

 INFO  Discovering packages.

laravel/sail ..... DONE
laravel/sanctum ..... DONE
laravel/tinker ..... DONE
nesbot/carbon ..... DONE
nunomaduro/collision ..... DONE
nunomaduro/termwind ..... DONE
spatie/laravel-ignition ..... DONE
```

Después de instalar el paquete Laravel Sanctum, es necesario ejecutar también este otro comando:

### ❖ sail artisan vendor:publish --provider="Laravel\Sanctum\SanctumServiceProvider"

Esto sirve para publicar (copiar) los archivos y configuraciones necesarios de Sanctum a la aplicación. Por ejemplo, puede incluir la publicación de migraciones, configuraciones, vistas y otros archivos necesarios para que Sanctum funcione correctamente en el proyecto Laravel.

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$
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
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$
```

## 4.2 Creación de AuthController

Una vez hecho lo anterior, ahora podemos crear un nuevo controlador, en este caso “AuthController” que será de utilidad para definir las funciones de **login**, **registro** y **logout**.

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/ApiTareas$
sail artisan make:controller AuthController

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

Es importante saber que este controlador va a utilizar el modelo **User**, que forma parte de los archivos que Laravel define automáticamente cuando creamos un nuevo proyecto. Este modelo incluye la lógica necesaria para manejar la autenticación de usuarios, como el registro, el inicio de sesión y el cierre de sesión. Se relaciona con la tabla users en la base de datos y es utilizado por Laravel Sanctum para gestionar la autenticación y emisión de tokens de acceso.

```
use App\Models\User;
```

### Función register() para registrar un nuevo usuario (POST):

```
public function register(Request $request){
    //Crear un nuevo usuario en la base de datos usando el modelo User
    $user = User::create([
        'name'=>$request->name,
        'email'=>$request->email,
        //hashear la contraseña para que se guarde de forma segura
        'password'=>Hash::make($request->password)
    ]);
    //Crear un token de acceso para el usuario recién registrado
    $token = $user->createToken('auth_token')->plainTextToken;

    //Devuelve una respuesta json con los detalles del usuario y el token de acceso
    return response()->json(['data'=>$user, 'access_token'=>$token,
        'token_type'=>'Bearer']); //tipo de autenticacion
}
```

---

\***plainTextToken** → se utiliza en Laravel Sanctum para obtener el token de acceso en texto plano, lo que significa que el token no está encriptado ni cifrado, y su valor puede ser leído directamente.

---

### Función login() para iniciar sesión con un usuario (POST):

```
public function login(Request $request){
    // Buscar al usuario en la base de datos por su dirección de correo electrónico
    $user = User::where('email', $request->email)->firstOrFail();

    // Verificar si la contraseña proporcionada y hasheada no coincide con la contraseña
    //almacenada en la base de datos
    if(!Hash::check($request->password, $user->password)){
        //devuelve mensaje de error en json si no coinciden
        return response()->json(['message'=> 'Credenciales incorrectas'], 401);
    }

    //crear un nuevo token de acceso para el usuario autenticado
    $token = $user->createToken('auth_token')->plainTextToken;

    //Devuelve respuesta json con los detalles de usuario y un mensaje de saludo
    return response()->json(['message'=>'Hola ' . $user->name,
        'access_token'=>$token,
        'token_type'=>'Bearer']);
}
```

### Función logout() cerrar sesión de un usuario (GET):

```
public function logout(){
    // Revocar y eliminar todos los tokens de acceso asociados al usuario autenticado
    auth()->user()->tokens()->delete();
    // Devolver un mensaje de sesión cerrada
    return ['message' =>'Sesión cerrada correctamente'];
}
```

**auth()->user()->tokens()->delete():**

- ❖ **auth()->user()** devuelve la instancia del usuario autenticado actualmente.
- ❖ **tokens()** devuelve una colección de todos los tokens de acceso asociados a ese usuario.
- ❖ **delete()** se utiliza para eliminar esos tokens.

## 4.3 Rutas protegidas por Sanctum

Una vez tenemos listas las funciones, tenemos que realizar una modificación en api.php para verificar el correcto funcionamiento de la autenticación de usuarios, además de especificar que el archivo utilice también AuthController .

```
use App\Http\Controllers\AuthController;
```

El código siguiente explica cómo debe de estar estructurado:

```
//Definir las solicitudes post para el registro y el login de usuarios
Route::post('register', [AuthController::class, 'register']);
Route::post('login', [AuthController::class, 'login']);

// Rutas protegidas que requieren autenticación mediante Sanctum
Route::middleware('auth:sanctum')->group(function(){
    //Cierre de sesión con get
    Route::get('logout', [AuthController::class, 'logout']);
    //Recursos de tareas y etiquetas
    Route::resource('/tareas', TareaController::class);
    Route::resource('/etiquetas', EtiquetaController::class);
});
```

---

**Route::middleware('auth:sanctum')->group(function()** → recoge todas las funciones protegidas por Sanctum, por lo que solo se puede acceder a ellas si se ha autenticado un usuario. Esto hace que solo se pueda interactuar con las tareas y etiquetas en el momento en el que un usuario esté logueado, y también es la única forma de hacer el cierre de sesión.

---

## 5. Creación de los tests

En Laravel, los tests son un componente esencial para garantizar la integridad y el correcto funcionamiento de una aplicación, dado que son fragmentos de código diseñados para verificar que cada componente de la aplicación funciona como se espera.

Sabiendo esto, vamos a crear tests para comprobar que las tareas, las etiquetas y la autenticación de usuario funcionan correctamente. Se utiliza el siguiente comando para crearlo:

❖ **sail artisan make:test <nombre\_test>**

**Creación de los tests AuthTest, TareasTest, EtiquetasTest:**

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/APItareas$
sail artisan make:test AuthTest

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



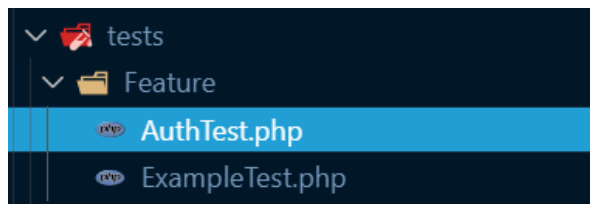
```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/APItareas$
sail artisan make:test TareasTest

INFO Test [tests/Feature/TareasTest.php] created successfully.

irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/APItareas$
sail artisan make:test EtiquetasTest

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

Estos tests se encuentran almacenados en la carpeta “tests”, dentro de “Feature”:



---

\* Es importante definir dentro de los tests el uso de **RefreshDatabase**, ya que este trait sirve para **resetear la base de datos** después de cada test y evitar cualquier error en esta.

---

De los tests que se han creado, se pondrá como ejemplo **AuthTest** para analizar el código que lo define.

Dentro de AuthTest vamos a hacer comprobaciones para saber si las funciones de registro, login y logout funcionan adecuadamente. Para ello se utilizarán funciones que harán simulaciones de prueba para hacer las comprobaciones necesarias. A continuación, se mostrará una explicación del código de cada función:

### Función testRegister → comprobar el registro:

La siguiente función crea como simulación un nuevo usuario utilizando un array para definir varias propiedades, en este caso name, email y password (la cual está hasheada para que sea más segura). Después se envía una solicitud con post en formato json al endpoint **/api/register** utilizando los datos de prueba del usuario de simulación y por último se utiliza un **assertStatus(200)** para verificar que se responda de forma exitosa con un código de estado **HTTP 200 (éxito)** y también se asegura con **assertJsonStructure()** de que la respuesta en json siga una estructura específica (debe tener las claves ‘data’, ‘access\_token’ y ‘token\_type’).

```
public function testRegister()
{
    // Crear un usuario de ejemplo en la base de datos para registrarse
    $usuarioTest = [
        'name' => 'Test',
        'email' => 'Test@test.com',
        'password' => Hash::make('secret'),
    ];

    // Enviar una solicitud POST JSON a la ruta '/api/register' con los datos de registro
    $response = $this->postJson('/api/register', $usuarioTest);

    // Asegurarse de que la respuesta tenga un código de estado 200 (éxito)
    $response->assertStatus(200)
    //Asegurarse de que la estructura del json es la siguiente
    ->assertJsonStructure([
        'data',
        'access_token',
        'token_type',
    ]);
}
```

### Función testLogin → comprobar el inicio de sesión:

En esta función se mantiene una estructura parecida a la de registro, salvo que en este caso primero se añade un usuario de simulación a la base de datos y luego se mandan los mismos datos de este usuario como request para realizar la solicitud al endpoint **/api/login** que comprobará si los datos coinciden. También se comprobará que en la respuesta en json a la solicitud aparezca el mensaje de saludo al usuario logueado y el tipo de token (**Bearer**).



```
public function testLogin()
{
    // Crear un usuario de ejemplo en la base de datos para iniciar sesión
    $user = User::factory()->create([
        'email' => 'Test@test.com',
        'password' => Hash::make('secret'),
    ]);

    // Datos de inicio de sesión para el usuario para ser enviados
    $usuarioTest = [
        'email' => $user->email,
        'password' => 'secret',
    ];

    // Enviar una solicitud POST JSON a la ruta '/api/login' con los datos de inicio de sesión
    $response = $this->postJson('/api/login', $usuarioTest);

    // Asegurarse de que la respuesta tenga un código de estado 200 (éxito)
    $response->assertStatus(200)
    // Asegurarse de que la respuesta contenga ciertos elementos en formato JSON
    ->assertJson([
        'message' => 'Hola ' . $user->name,
        'token_type' => 'Bearer',
    ]);
}
```

### Función testLogout() → comprobar el cierre de sesión:

En este caso realizamos la simulación de que tenemos un usuario en la base de datos y le creamos un token de acceso para poder enviarlo en las cabeceras al endpoint /api/logout utilizando withHeaders, comprobando que dé como respuesta un código de estado 200 para garantizar el éxito y el mensaje “sesión cerrada correctamente”.

```
public function testLogout()
{
    // Crear un usuario de ejemplo en la base de datos
    $user = User::factory()->create();
    // Crear un token de acceso para el usuario
    $token = $user->createToken('auth_token')->plainTextToken;

    // Enviar una solicitud JSON con el token de acceso para cerrar la sesión
    $response = $this->withHeaders(['Authorization' => 'Bearer ' . $token])
        ->json('GET', '/api/logout');

    // Asegurarse de que la respuesta tenga un código de estado 200 (éxito)
    $response->assertStatus(200)
        ->assertJson(['message' => 'Sesión cerrada correctamente']);
}
```

Por último para poner en ejecución los test, se utiliza el siguiente comando:

### ❖ sail artisan test

Si los tests han sido exitosos aparecerá un resultado como el siguiente:

```
irene@LAPTOP-A0EUTHC8:/mnt/c/Users/Usuario/Desktop/2DAW/Desarrollo web en entorno servidor/Docker_compose/www/APIareas$
sail artisan test

PASS Tests\Unit\ExampleTest
✓ that true is true 0.13s

PASS Tests\Feature\AuthTest
✓ register 6.13s
✓ login 0.38s
✓ logout 0.13s

PASS Tests\Feature\EtiquetasTest
✓ listado etiquetas 0.14s
✓ mostrar etiqueta 0.05s
✓ borrar etiqueta 0.05s

PASS Tests\Feature\ExampleTest
✓ the application returns a successful response 0.36s

PASS Tests\Feature\TareasTest
✓ listado tareas 0.07s
✓ mostrar tarea 0.08s
✓ borrar tarea 0.05s

Tests: 11 passed (30 assertions)
Duration: 8.98s
```

## 6. Comprobación en postman

A continuación, se utilizará la aplicación postman para probar la API creada. Esta aplicación permitirá realizar las peticiones HTTP explicadas anteriormente para comprobar el funcionamiento del proyecto.

### 6.1 comprobación de registro, login y logout

#### Registro de un nuevo usuario (función register()):

Como tenemos que enviar datos a la solicitud, hay que ir al apartado Body>raw.

# Instalación de un servidor ubuntu en AWS

POST http://localhost/api/register

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Beautify

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "name": "TestIrene",
3   "email": "TestIrene@gmail.com",
4   "password": "12345"
5 }
```

Body Cookies Headers (9) Test Results Status: 200 OK Time: 3.70 s Size: 538 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "data": {
3     "name": "TestIrene",
4     "email": "TestIrene@gmail.com",
5     "updated_at": "2024-03-03T17:04:29.000000Z",
6     "created_at": "2024-03-03T17:04:29.000000Z",
7     "id": 2
8   },
9   "access_token": "3|fHhRxYYfZk0DUVpAKtnjLx8oFglagPwAIHCckp175593694",
10  "token_type": "Bearer"
11 }
```

## Inicio de sesión del nuevo usuario (función login()):

POST http://localhost/api/login

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Beautify

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "email": "TestIrene@gmail.com",
3   "password": "12345"
4 }
```

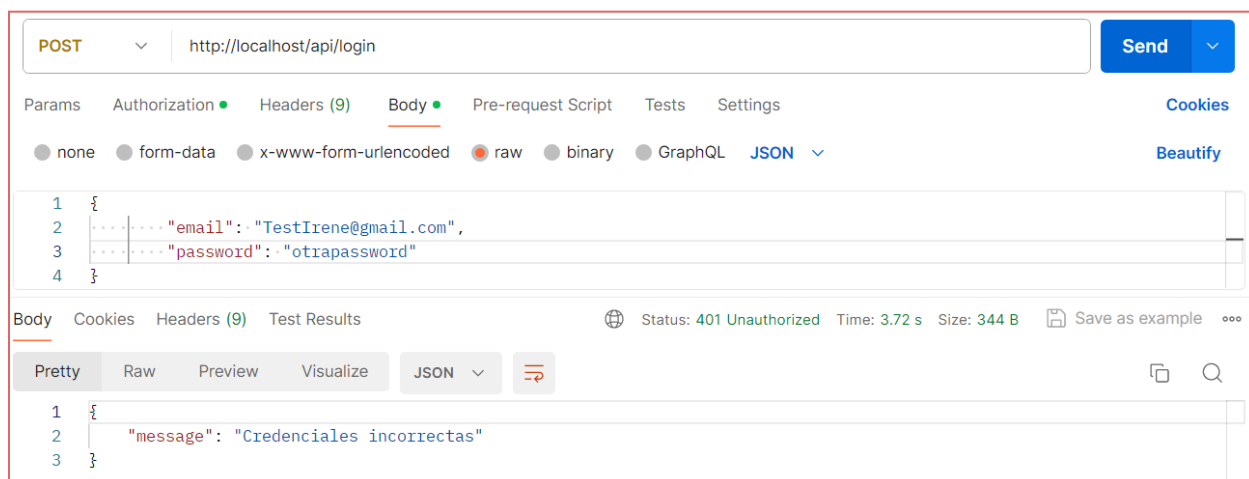
Body Cookies Headers (9) Test Results Status: 200 OK Time: 3.62 s Size: 414 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "message": "Hola TestIrene",
3   "access_token": "4|PIpjKuGpgu4sit18SlECQPxqhc4PstIBQ3TS1gVF754c3541",
4   "token_type": "Bearer"
5 }
```

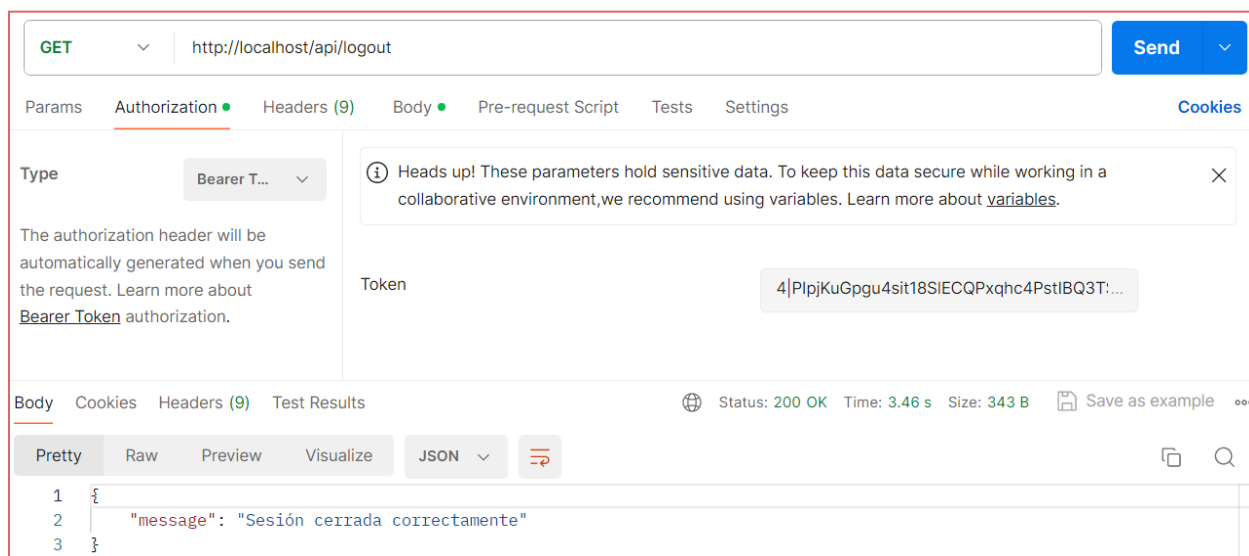
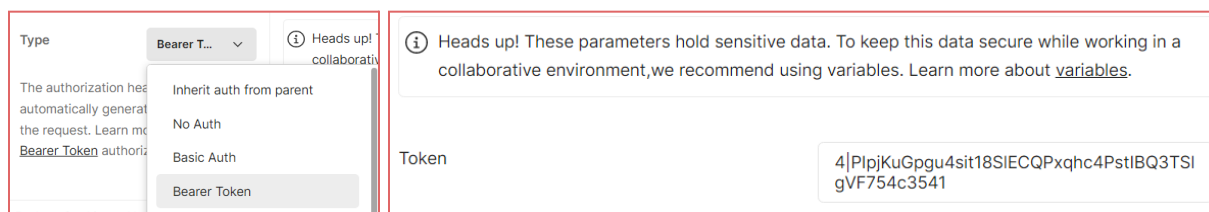
En el caso de que se haya puesto algún dato incorrecto aparecerá el siguiente mensaje:

# Instalación de un servidor ubuntu en AWS



## Cierre de sesión del nuevo usuario (función logout()):

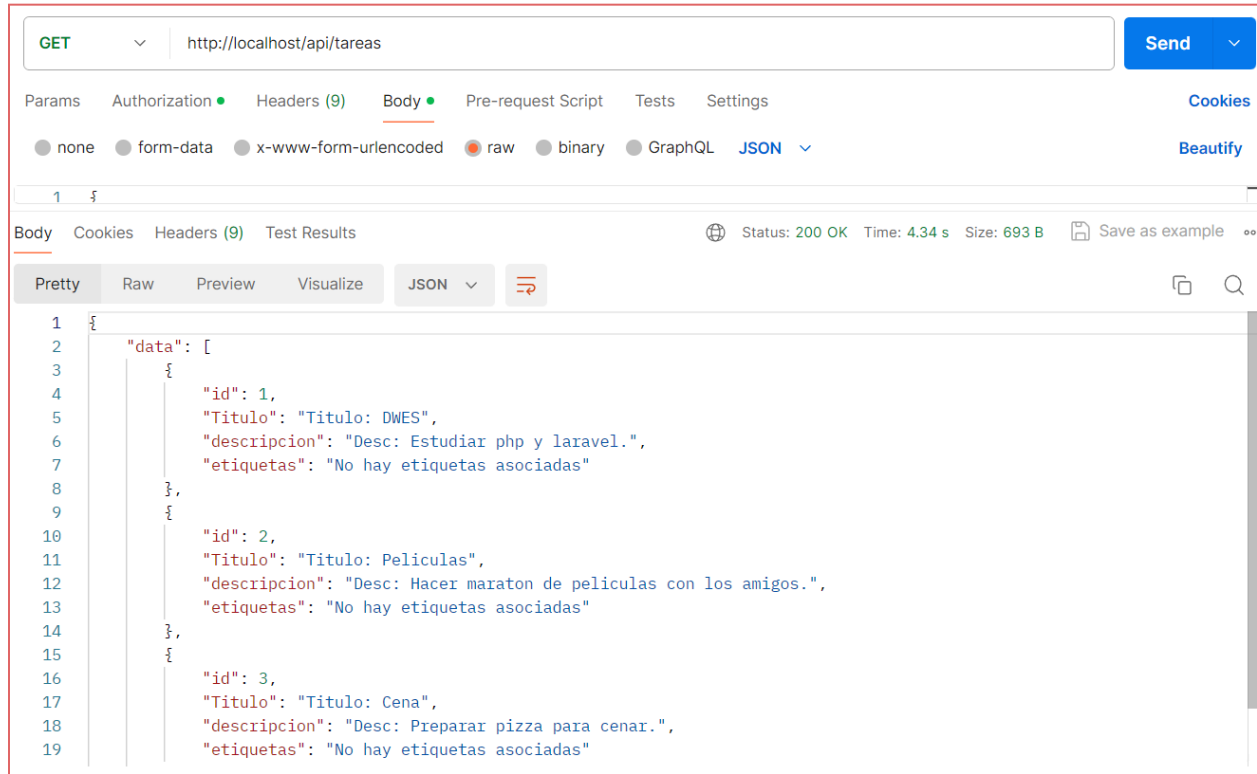
Para esta función tenemos que copiar y pegar el **access\_token** que obtenemos en la respuesta del login e irmo al apartado “**Authorization**”. Una vez ahí, hay que pegar el token en su apartado y seleccionar en el tipo (Type) “**Bearer Token**”.



Hay que tener en cuenta que como restringimos el acceso a las tareas y a las etiquetas, es necesario actualizar el token en el apartado de Authorization para poder acceder.

## 6.2 comprobación de tareas

### Listado de todas las tareas (función index()):



```
GET http://localhost/api/tareas

Body
  none form-data x-www-form-urlencoded raw binary GraphQL JSON
  1 {
  2   "data": [
  3     {
  4       "id": 1,
  5       "Titulo": "Titulo: DWES",
  6       "descripcion": "Desc: Estudiar php y laravel.",
  7       "etiquetas": "No hay etiquetas asociadas"
  8     },
  9     {
 10       "id": 2,
 11       "Titulo": "Titulo: Peliculas",
 12       "descripcion": "Desc: Hacer maraton de peliculas con los amigos.",
 13       "etiquetas": "No hay etiquetas asociadas"
 14     },
 15     {
 16       "id": 3,
 17       "Titulo": "Titulo: Cena",
 18       "descripcion": "Desc: Preparar pizza para cenar.",
 19       "etiquetas": "No hay etiquetas asociadas"
 20     }
 21   ]
 22 }
```

### Creación de una nueva tarea (función store()):

En esta nueva tarea también se añadirá una relación con 2 etiquetas, para ello especificamos en el json un array con los ids de las etiquetas existentes que queremos añadir.



# Instalación de un servidor ubuntu en AWS

POST http://localhost/api/tareas

Params Authorization Headers (10) Body Pre-request Script Tests Settings Cookies Beautify

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "id": 5,
3   "titulo": "prueba",
4   "descripcion": "prueba para crear nueva tarea",
5   "etiquetas": [1,2]
6 }
```

Body Cookies (2) Headers (9) Test Results Status: 201 Created Time: 3.73 s Size: 427 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "data": {
3     "id": 5,
4     "Titulo": "Titulo: prueba",
5     "descripcion": "Desc: prueba para crear nueva tarea",
6     "etiquetas": [
7       "Estudio",
8       "Ocio"
9     ]
10  }
11 }
```

## Mostrar una tarea específica según su id (función show()):

GET http://localhost/api/tareas/2

Params Authorization Headers (10) Body Pre-request Script Tests Settings Cookies Beautify

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
```

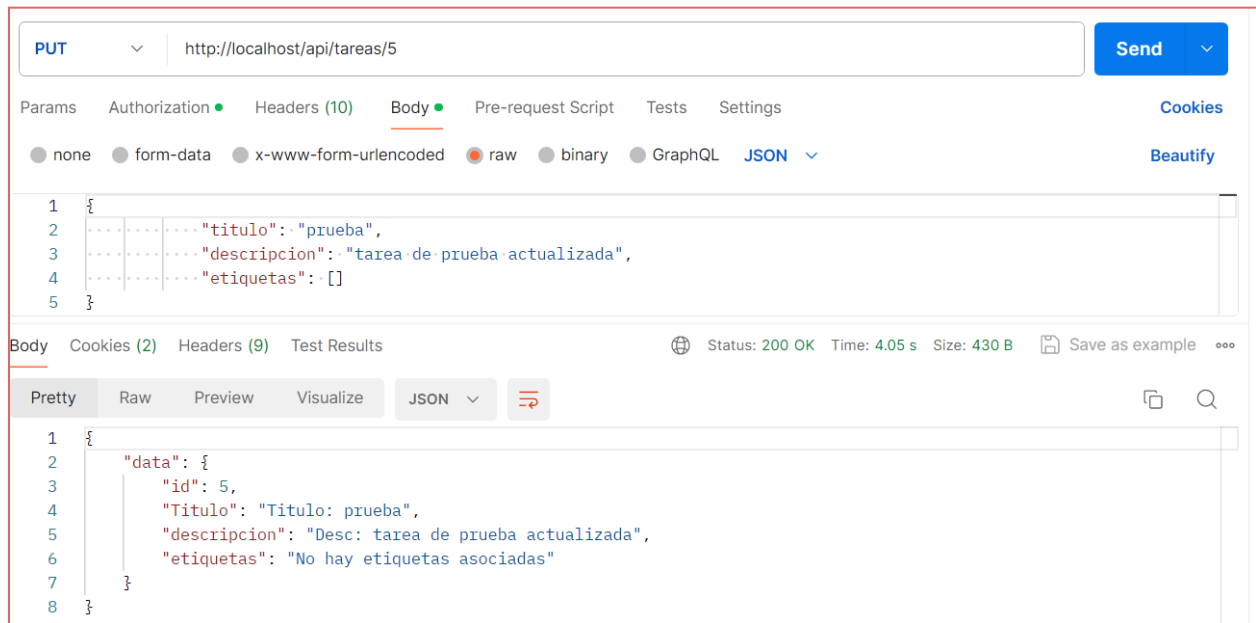
Body Cookies (2) Headers (9) Test Results Status: 200 OK Time: 3.69 s Size: 448 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "data": {
3     "id": 2,
4     "Titulo": "Titulo: Peliculas",
5     "descripcion": "Desc: Hacer maraton de peliculas con los amigos.",
6     "etiquetas": "No hay etiquetas asociadas"
7   }
8 }
```



## Modificar una tarea existente (función update()):



PUT ▼ http://localhost/api/tareas/5 Send ▼

Params Authorization ● Headers (10) **Body** ● Pre-request Script Tests Settings Cookies

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL JSON ▼ Beautify

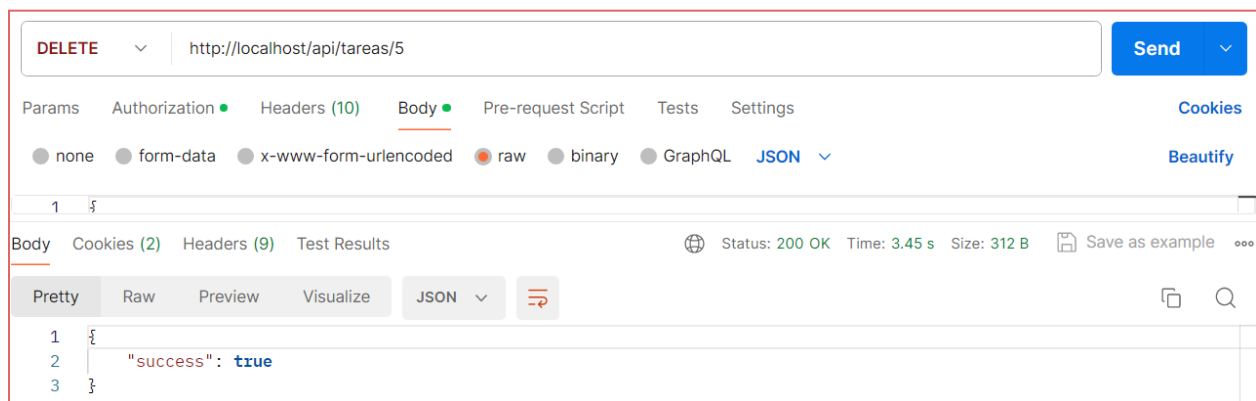
```
1 {
2   ..... "titulo": "prueba",
3   ..... "descripcion": "tarea de prueba actualizada",
4   ..... "etiquetas": []
5 }
```

Body Cookies (2) Headers (9) Test Results 🌐 Status: 200 OK Time: 4.05 s Size: 430 B 📄 Save as example ⋮

Pretty Raw Preview Visualize JSON ▼ 🔍

```
1 {
2   "data": {
3     "id": 5,
4     "Titulo": "Titulo: prueba",
5     "descripcion": "Desc: tarea de prueba actualizada",
6     "etiquetas": "No hay etiquetas asociadas"
7   }
8 }
```

## Borrar una tarea (función destroy()):



DELETE ▼ http://localhost/api/tareas/5 Send ▼

Params Authorization ● Headers (10) **Body** ● Pre-request Script Tests Settings Cookies

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL JSON ▼ Beautify

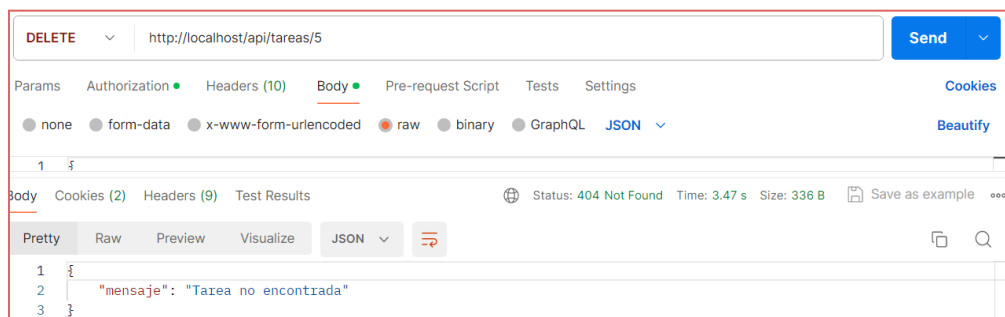
```
1 {
```

Body Cookies (2) Headers (9) Test Results 🌐 Status: 200 OK Time: 3.45 s Size: 312 B 📄 Save as example ⋮

Pretty Raw Preview Visualize JSON ▼ 🔍

```
1 {
2   "success": true
3 }
```

Ahora podemos ver un ejemplo de qué ocurriría si indicamos el id de la tarea que acabamos de borrar.



DELETE ▼ http://localhost/api/tareas/5 Send ▼

Params Authorization ● Headers (10) **Body** ● Pre-request Script Tests Settings Cookies

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL JSON ▼ Beautify

```
1 {
```

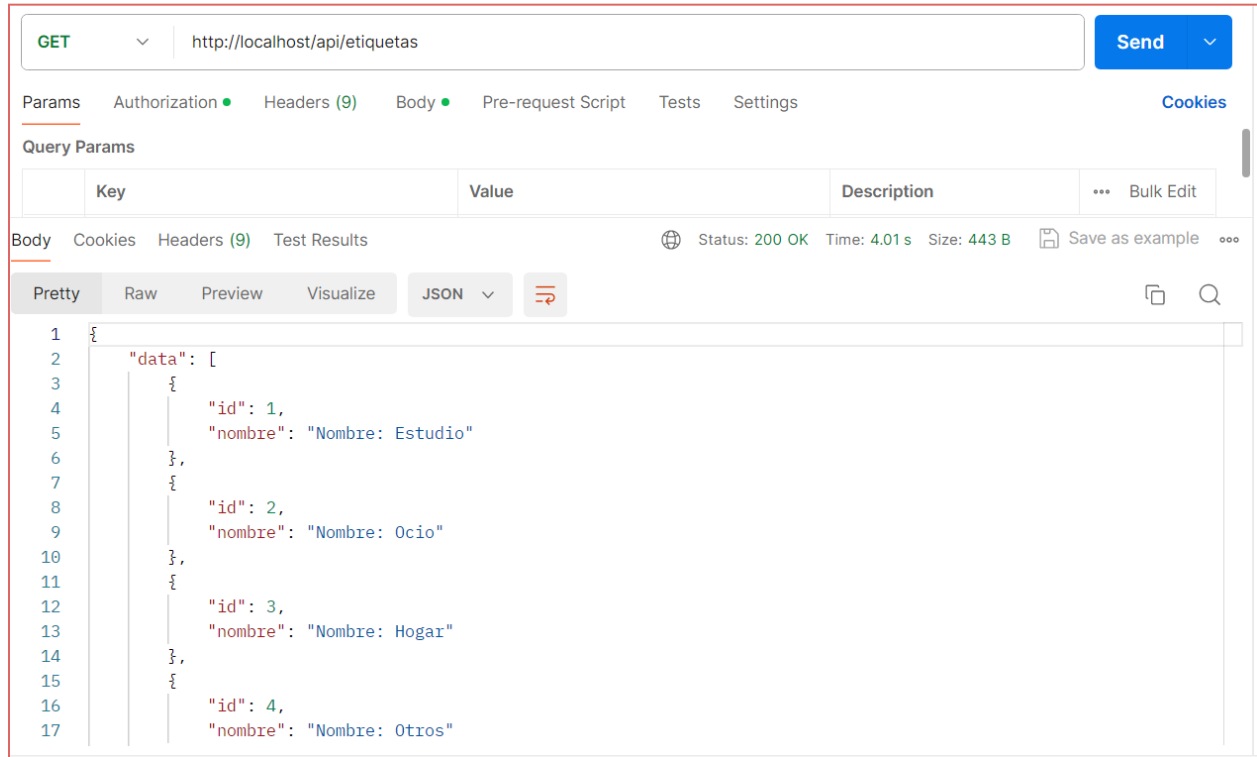
Body Cookies (2) Headers (9) Test Results 🌐 Status: 404 Not Found Time: 3.47 s Size: 336 B 📄 Save as example ⋮

Pretty Raw Preview Visualize JSON ▼ 🔍

```
1 {
2   "mensaje": "Tarea no encontrada"
3 }
```

## 6.3 comprobación de etiquetas

### Listado de todas las etiquetas (función index()):



GET `http://localhost/api/etiquetas` Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

Query Params

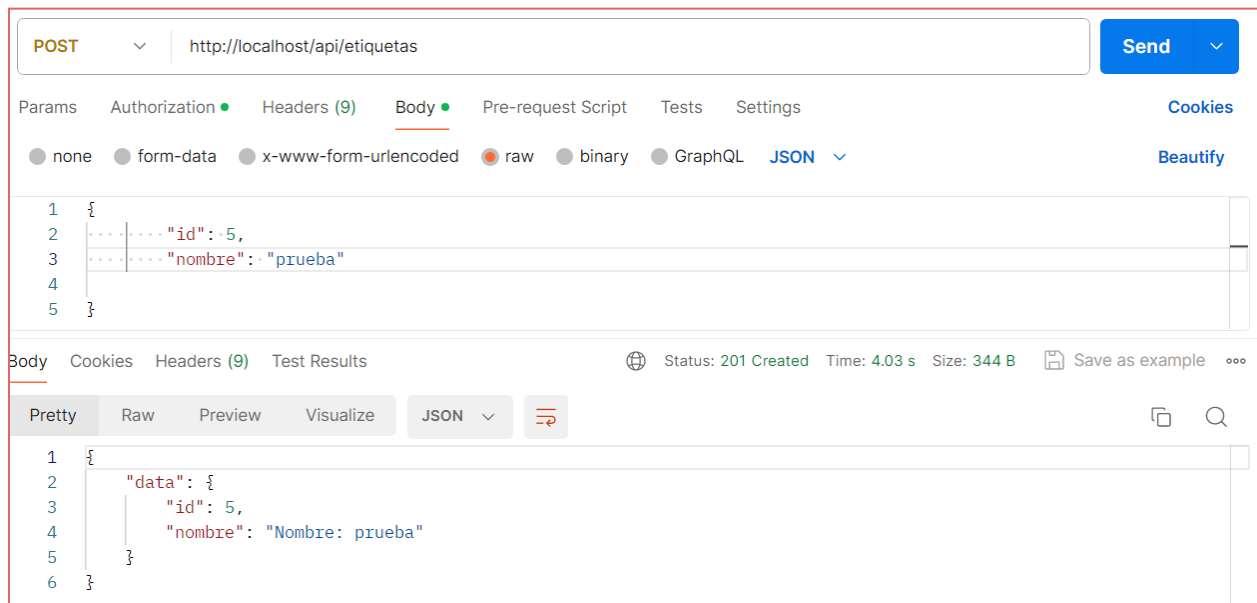
| Key | Value | Description | Bulk Edit |
|-----|-------|-------------|-----------|
|-----|-------|-------------|-----------|

Body Cookies Headers (9) Test Results Status: 200 OK Time: 4.01 s Size: 443 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "data": [
3     {
4       "id": 1,
5       "nombre": "Nombre: Estudio"
6     },
7     {
8       "id": 2,
9       "nombre": "Nombre: Ocio"
10    },
11    {
12      "id": 3,
13      "nombre": "Nombre: Hogar"
14    },
15    {
16      "id": 4,
17      "nombre": "Nombre: Otros"
```

### Creación de una nueva etiqueta (función store()):



POST `http://localhost/api/etiquetas` Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

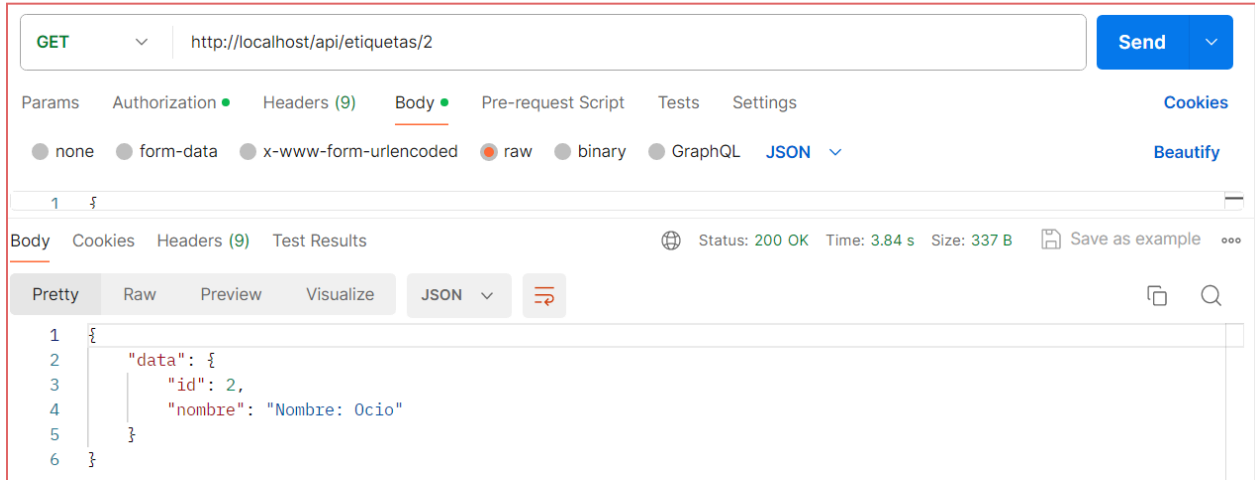
```
1 {
2   "id": 5,
3   "nombre": "prueba"
4 }
5 }
```

Body Cookies Headers (9) Test Results Status: 201 Created Time: 4.03 s Size: 344 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "data": {
3     "id": 5,
4     "nombre": "Nombre: prueba"
5   }
6 }
```

## Mostrar una etiqueta específica según su id (función show()):



GET ▼ http://localhost/api/etiquetas/2 Send ▼

Params Authorization ● Headers (9) **Body** ● Pre-request Script Tests Settings Cookies Beautify

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL **JSON** ▼

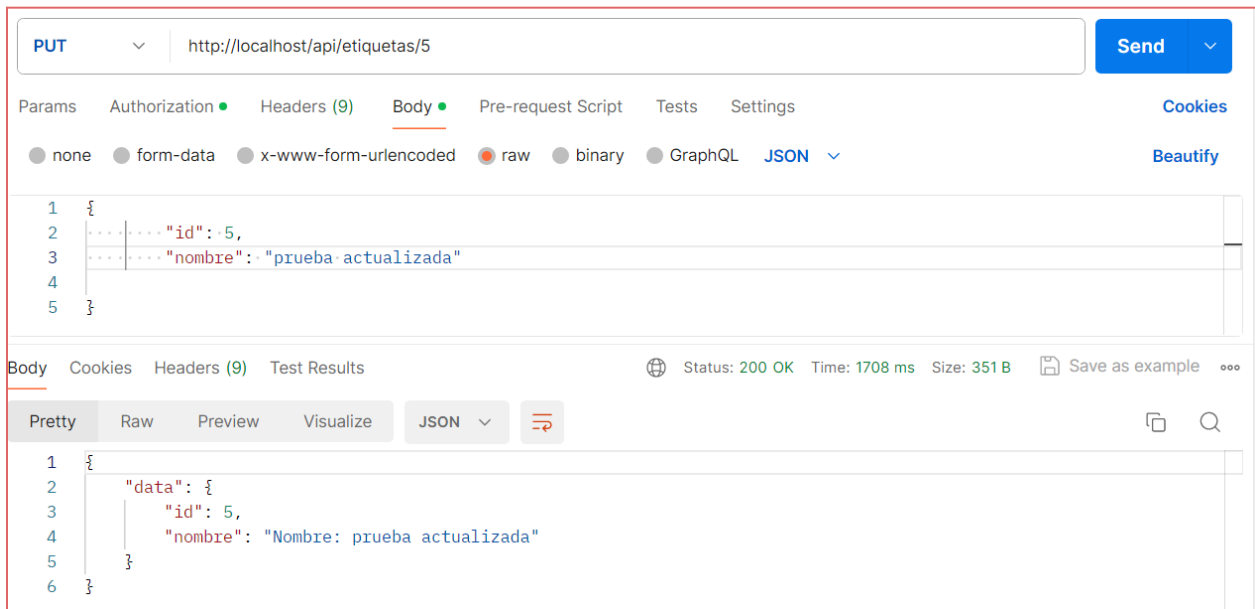
1 {

Body Cookies Headers (9) Test Results 🌐 Status: 200 OK Time: 3.84 s Size: 337 B 📄 Save as example ⋮

Pretty Raw Preview Visualize **JSON** ▼ 🔍

```
1 {
2   "data": {
3     "id": 2,
4     "nombre": "Nombre: Ocio"
5   }
6 }
```

## Modificar una etiqueta existente (función update()):



PUT ▼ http://localhost/api/etiquetas/5 Send ▼

Params Authorization ● Headers (9) **Body** ● Pre-request Script Tests Settings Cookies Beautify

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL **JSON** ▼

```
1 {
2   "id": 5,
3   "nombre": "prueba actualizada"
4 }
5
```

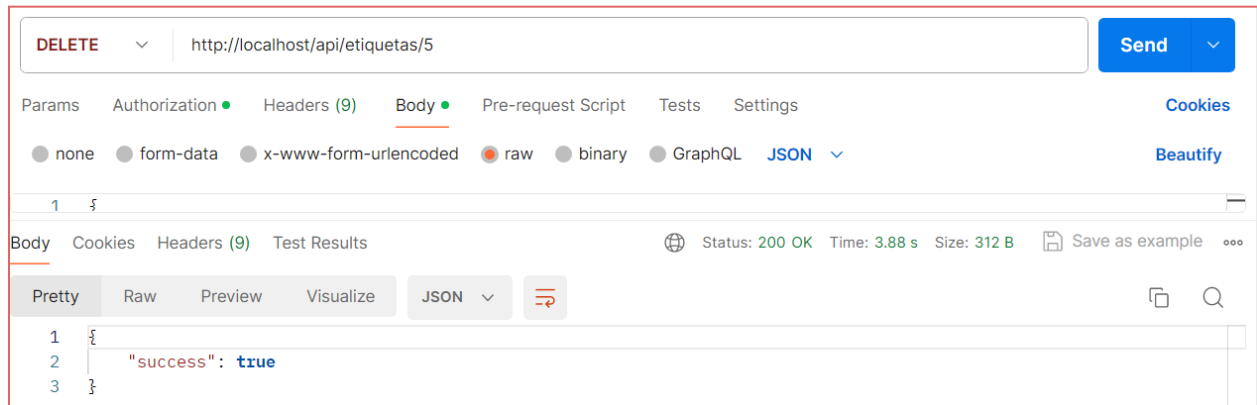
Body Cookies Headers (9) Test Results 🌐 Status: 200 OK Time: 1708 ms Size: 351 B 📄 Save as example ⋮

Pretty Raw Preview Visualize **JSON** ▼ 🔍

```
1 {
2   "data": {
3     "id": 5,
4     "nombre": "Nombre: prueba actualizada"
5   }
6 }
```



## Borrar una etiqueta (función destroy()):



Ahora podemos ver lo que ocurre si volvemos a realizar la petición de DELETE con el mismo id de la tarea que acaba de ser eliminada:

