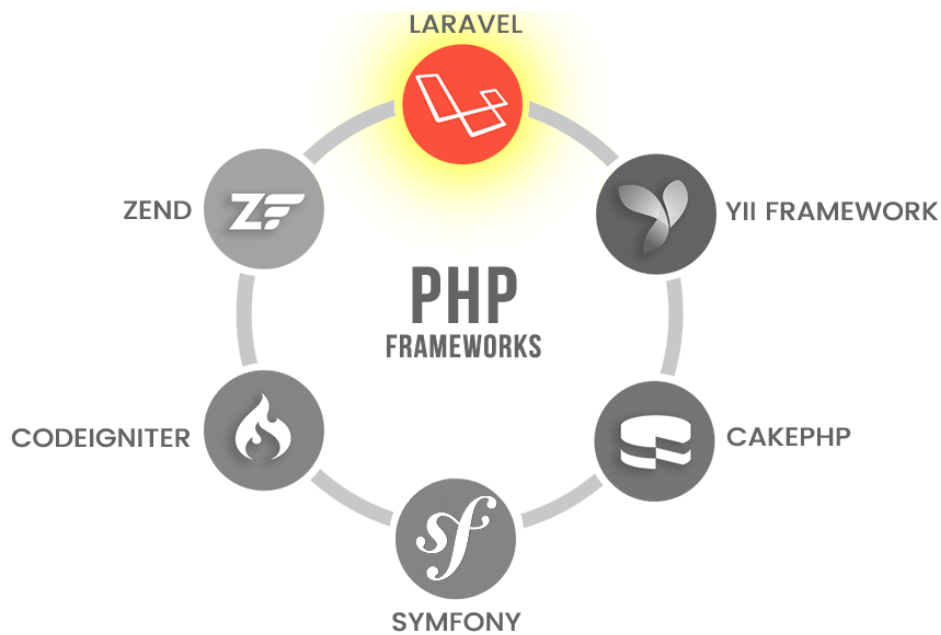


## unidad didáctica 8

# Laravel - servicios REST actividades



## 1. **ejemplo API Rest en tabla productos**

- 1. 1. [crear tabla productos](#)
- 1. 2. [crear controlador ProductoController](#)
- 1. 3. [cómo funciona la API REST](#)
  - 1. 3. 1. [listar todos los productos](#)
  - 1. 3. 2. [listar un producto en concreto](#)
  - 1. 3. 3. [introducir producto nuevo](#)
  - 1. 3. 4. [actualizar un producto existente](#)
  - 1. 3. 5. [eliminar un producto](#)

## 2. **ejercicios propuestos**

- 2. 1. [Ejercicio 1](#)

## 3. **bibliografía**

# 1. ejemplo API Rest en tabla productos

## 1.1. crear tabla productos

Antes de crear nuestra API en tabla `Productos` deberemos tener dicha tabla migrada en nuestro sistema. Para ello:

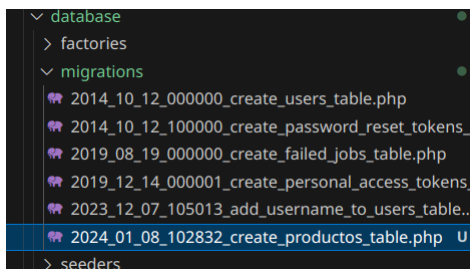
1. Crear **migración** para la tabla `productos`:

Recuerda que el nombre de la migración contiene palabras reservadas para como son *create* y *table*.

```
1 php artisan make:migration create_productos_table
2 # ó, si no funciona, probar:
3 # sudo docker-compose exec myapp php artisan make:migration
  create_productos_table
```

```
abc@jolly-wright:~/Escritorio/IES/DWES/projectes/pru-udemy$ sudo docker-compose exec myapp php artisan make:
migration create_productos_table
[sudo] password for abc:

INFO Migration [database/migrations/2024_01_08_102832_create_productos_table.php] created successfully.
```

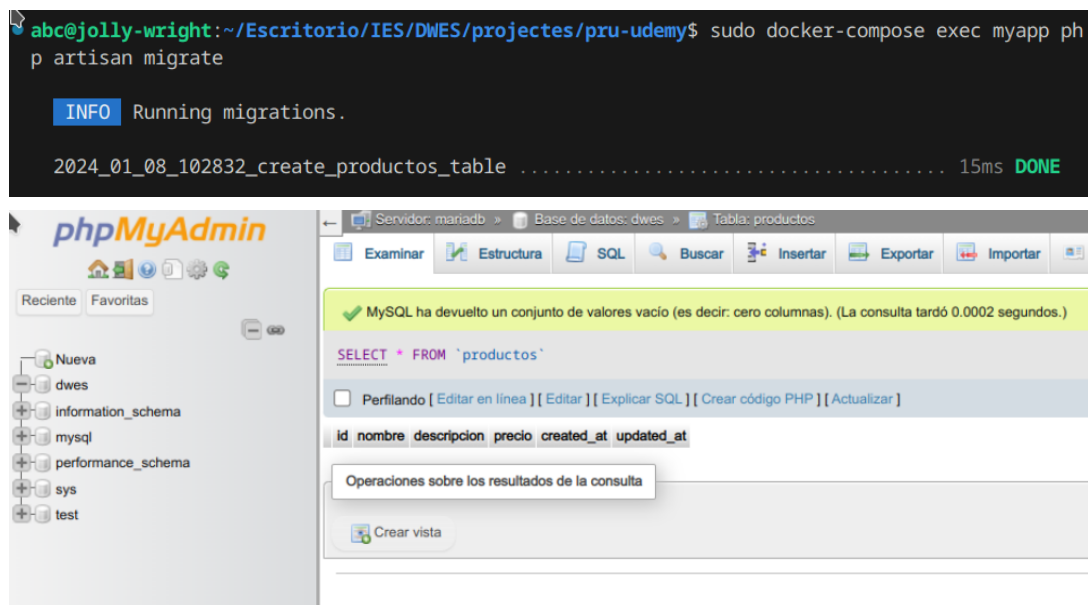


2. Añadir al fichero generado (en la carpeta `migrations` y en el ejemplo anterior `2024_01_08_102832_create_productos_table.php`) el resto de campos que se requieran en la tabla `productos`:

```
1 public function up(): void
2 {
3     Schema::create('productos', function (Blueprint $table) {
4         $table->id();
5         $table->string('nombre');
6         $table->text('descripcion');
7         $table->decimal('precio', 8, 2);
8         $table->timestamps();
9     });
10 }
```

2. Ejecutar migración:

```
1 php artisan migrate
2 # ó, si no funciona, probar:
3 # sudo docker-compose exec myapp php artisan migrate
```



3. Crear un `seeder` para realizar una carga de datos:

Introducimos información en esta tabla nueva, creando un fichero en la carpeta `database/seeder`s de nombre `ProductoSeeder.php`:

```

1  <?php
2      namespace Database\Seeders;
3      use Illuminate\Database\Seeder;
4      use Illuminate\Support\Facades\DB;
5
6      class ProductoSeeder extends Seeder {
7
8          public function run() {
9              // insertar datos prueba
10             DB::table('productos')->insert([
11                 'nombre' => 'producto prueba 1',
12                 'descripcion' => 'esta es una descripción para el producto
prueba 1',
13                 'precio' => 19.99,
14             ]);
15
16             DB::table('productos')->insert([
17                 'nombre' => 'producto prueba 2',
18                 'descripcion' => 'esta es una descripción para el producto
prueba 2',
19                 'precio' => 29.99,
20             ]);
21         }
22     }

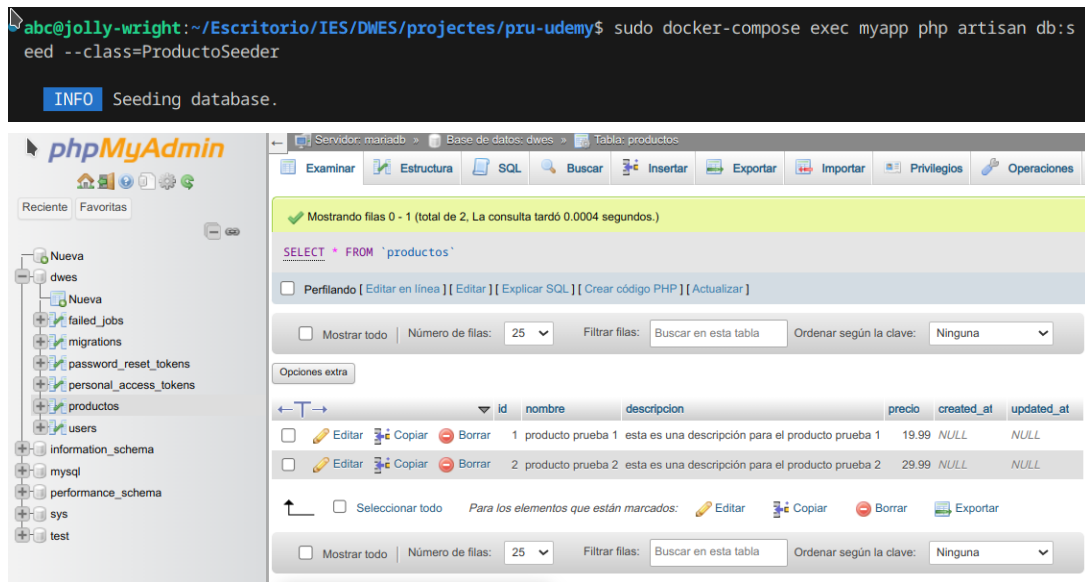
```

4. Ejecutar el `seeder`:

```

1  php artisan db:seed --class=ProductoSeeder
2  # ó, si no funciona, probar:
3  # sudo docker-compose exec myapp php artisan db:seed --
class=ProductoSeeder

```

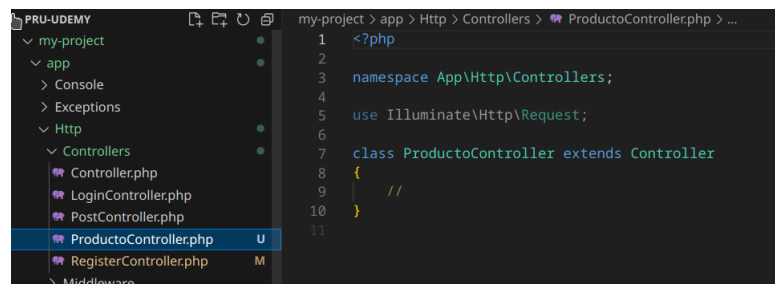
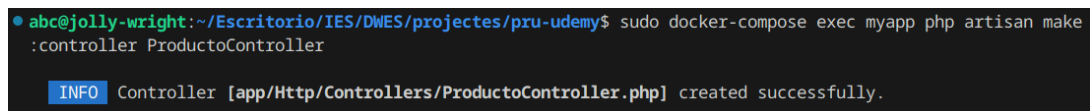


## 1.2. crear controlador ProductoController

Crear un controlador donde establezcamos los métodos que nosotros queramos realizar a la hora de trabajar con los datos.

1. **Crear** desde consola un controlador para la tabla `productos`:

```
1 php artisan make:controller ProductoController
2 # ó, si no funciona, probar:
3 # sudo docker-compose exec myapp php artisan make:controller
  ProductoController
```



La estructura de este archivo es un poco diferente a los controladores que ya hemos visto anteriormente. Ahora tenemos los siguientes métodos creados de manera automática:

- `index()` normalmente para listar (en nuestro caso los chollos).
- `create()` para crear plantillas (no lo vamos a usar).
- `store()` para guardar los datos que pasemos a la API.
- `update()` para actualizar un dato ya existente en la BD.
- `delete()` para eliminar un dato ya existente en la BD.

2. Como vamos a conectarnos a un modelo para traer la información de dicho modelo añadimos mediante `use`. También creamos la función `index` para listar todos los elementos de la tabla (en este caso `productos`):

```

1  <?php
2  namespace App\Http\Controllers;
3
4  use Illuminate\Http\Request;
5  use App\Models\Producto; // <-- esta linea
6
7  class ProductoController extends Controller
8  {
9      public function index(){
10         return response()->json(Producto::all());
11     }
12 }

```

**CUIDADO CON EL RETURN** porque ahora no estamos devolviendo una vista sino un array de datos en formato JSON.

3. Crear un modelo en la carpeta `Models` de nombre `Producto.php`:

```

1  <?php
2  namespace App\Models;
3
4  use Illuminate\Database\Eloquent\Model;
5
6  class Producto extends Model {
7      protected $fillable = ['nombre', 'descripcion', 'precio'];
8  }

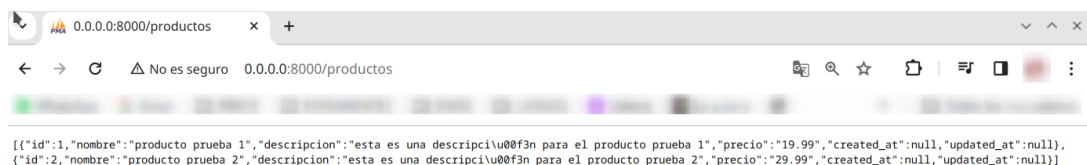
```

4. Ir a fichero `web.php` (en la carpeta `routes`) y colocar nuestras rutas:

```

1  // cargar el recurso del controlador ProductoController
2  use App\Http\Controllers\ProductoController
3
4
5  Route::prefix('productos')->group(function(){
6      Route::get('/', [ProductoController::class, 'index']);
7  });

```



```

[{"id":1,"nombre":"producto prueba 1","descripcion":"esta es una descripci\u00f3n para el producto prueba 1","precio":"19.99","created_at":null,"updated_at":null},
{"id":2,"nombre":"producto prueba 2","descripcion":"esta es una descripci\u00f3n para el producto prueba 2","precio":"29.99","created_at":null,"updated_at":null}]

```

La función anterior `index` nos devuelve todos los productos. Pero, qué pasa si queremos un producto en cuestión:

5. En `ProductoController.php` añadimos otra función (`show`) en la que se le pasa por parámetros el `id`:

```

1  <?php
2  namespace App\Http\Controllers;
3
4  use Illuminate\Http\Request;
5  use App\Models\Producto; // <-- esta linea
6

```

```

7  class ProductoController extends Controller
8  {
9      public function index(){
10         return response()->json(Producto::all());
11     }
12     public function show($id){
13         return response()->json(Producto::find($id));
14     }
15 }

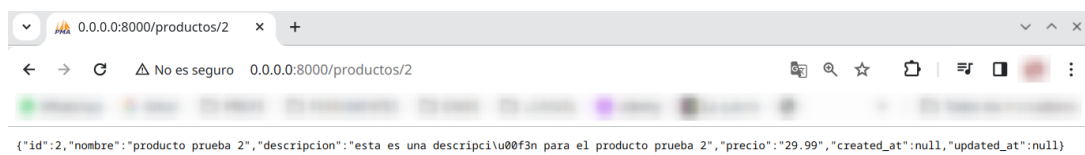
```

6. En `web.php` añadimos otra ruta en nuestro grupo:

```

1  Route::prefix('productos')->group(function(){
2      Route::get('/',[ProductoController::class, 'index']);
3      Route::get('/{id}',[ProductoController::class, 'show']);
4  });

```



7. Para introducir datos utilizaremos el método `store`:

a) en `ProductoController.php`:

```

1  public function store(Request $request){
2      $producto = Producto::create($request->all());
3      return response()->json($producto, 201);
4  }

```

b) en `web.php`:

```

1  Route::prefix('productos')->group(function(){
2      Route::get('/',[ProductoController::class, 'index']);
3      Route::get('/{id}',[ProductoController::class, 'show']);
4      Route::post('/',[ProductoController::class, 'store']);
5  });

```

8. Para actualizar datos de un producto, utilizaremos el método `update`:

a) en `ProductoController.php`:

```

1  public function update(Request $request, $id){
2      $producto = Producto::findOrFail($id);
3      $producto -> update($request->all());
4
5      return response()->json($producto, 200);
6  }

```

b) en `web.php`:

```
1 Route::prefix('productos')->group(function(){
2     Route::get('/',[ProductoController::class, 'index']);
3     Route::get('/{id}',[ProductoController::class, 'show']);
4     Route::post('/',[ProductoController::class, 'store']);
5     Route::put('/{id}',[ProductoController::class, 'update']);
6 });
```

9. Y para eliminar un producto, el método `delete`:

a) en `ProductoController.php`:

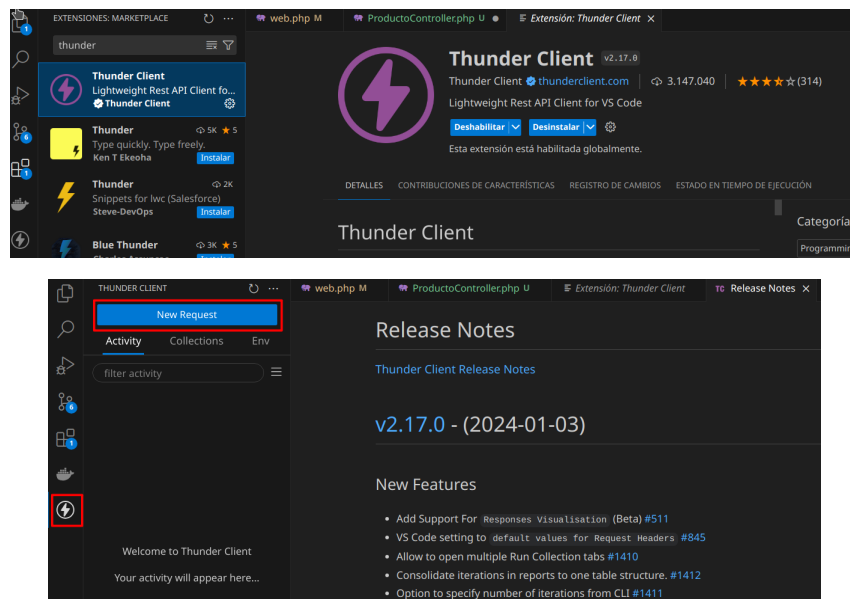
```
1 public function destroy($id){
2     Producto::findOrFail($id)->delete();
3
4     return response()->json(null, 204);
5 }
```

b) en `web.php`:

```
1 Route::prefix('productos')->group(function(){
2     Route::get('/',[ProductoController::class, 'index']);
3     Route::get('/{id}',[ProductoController::class, 'show']);
4     Route::post('/',[ProductoController::class, 'store']);
5     Route::put('/{id}',[ProductoController::class, 'update']);
6     Route::delete('/{id}',[ProductoController::class, 'destroy']);
7 });
```

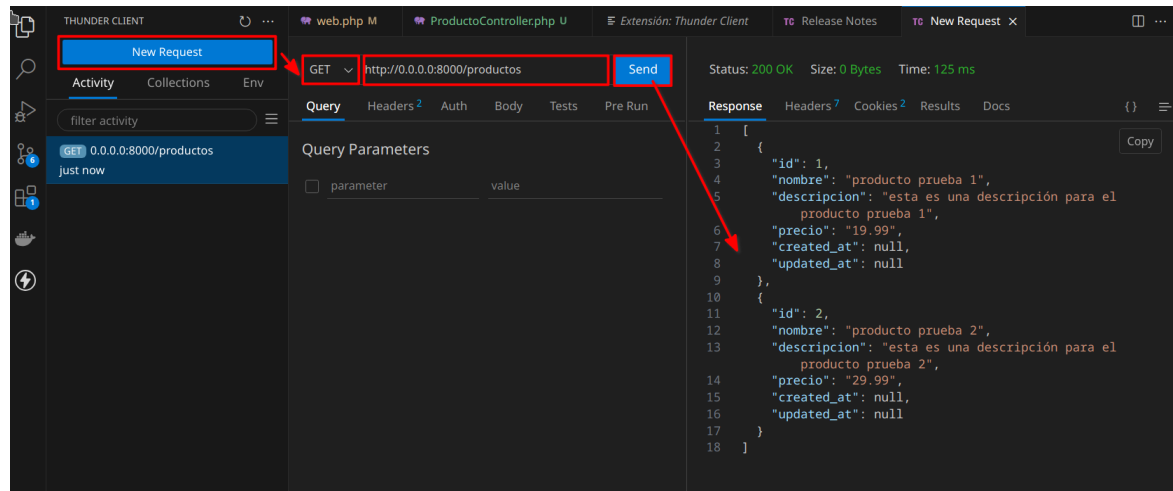
## 1.3. cómo funciona la API REST

Para ello vamos a utilizar un software que es una extensión de Visual Studio Code, de nombre `Thunder Client`:

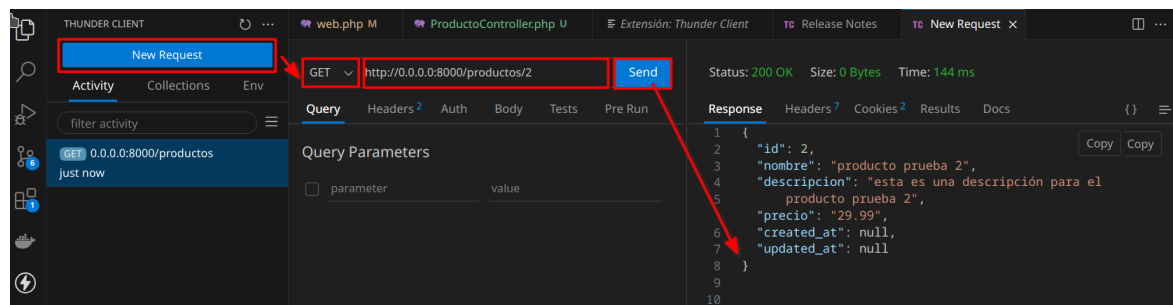




### 1.3.1. listar todos los productos



### 1.3.2. listar un producto en concreto



### 1.3.3. introducir producto nuevo

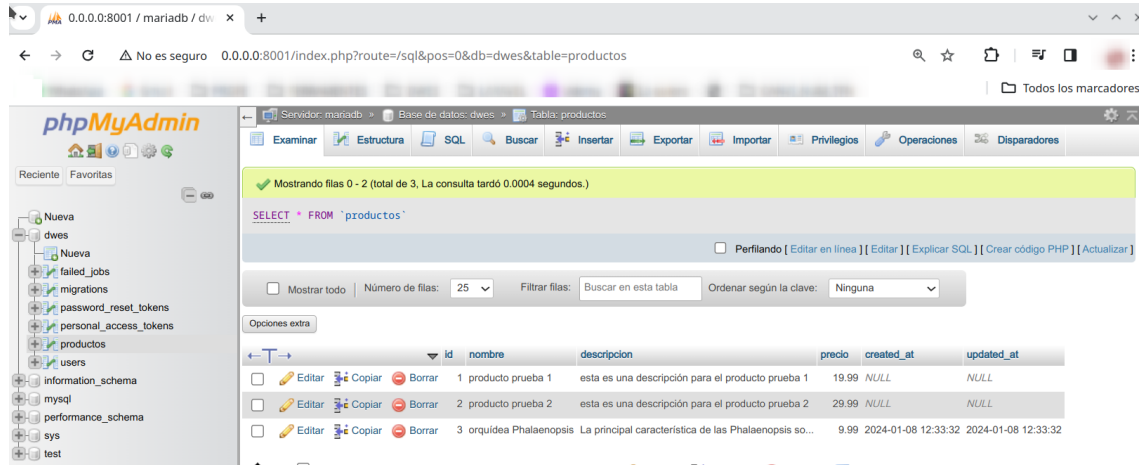
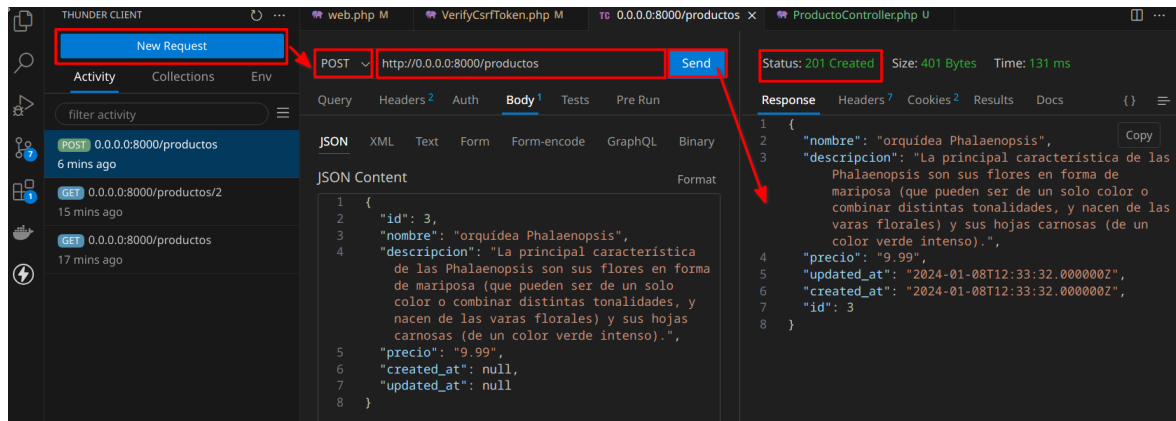
Si realizamos una nueva petición (new request) con método `post` y pasando (desde `body` y en `json`) un nuevo producto, va a mostrarnos un **error**.

Esto se debe a que Laravel, por sus métodos de seguridad, necesita un *token* llamado `csrf`.

Ya que, ahora mismo, estamos realizando pruebas, vamos a indicarle a Laravel que excluya la URL en cuestión de la verificación.

Para ello accedemos al fichero `VerifyCsrfToken.php` de la carpeta `app\Http\Middleware`:

```
1 <?php
2 namespace App\Http\Middleware;
3
4 use Illuminate\Foundation\Http\Middleware\VerifyCsrfToken as Middleware;
5
6 class VerifyCsrfToken extends Middleware
7 {
8     /**
9      * The URIs that should be excluded from CSRF verification.
10     *
11     * @var array<int, string>
12     */
13     protected $except = [
14         "http://0.0.0.0:8000/productos", // <-- esta excepción
15     ];
16 }
```



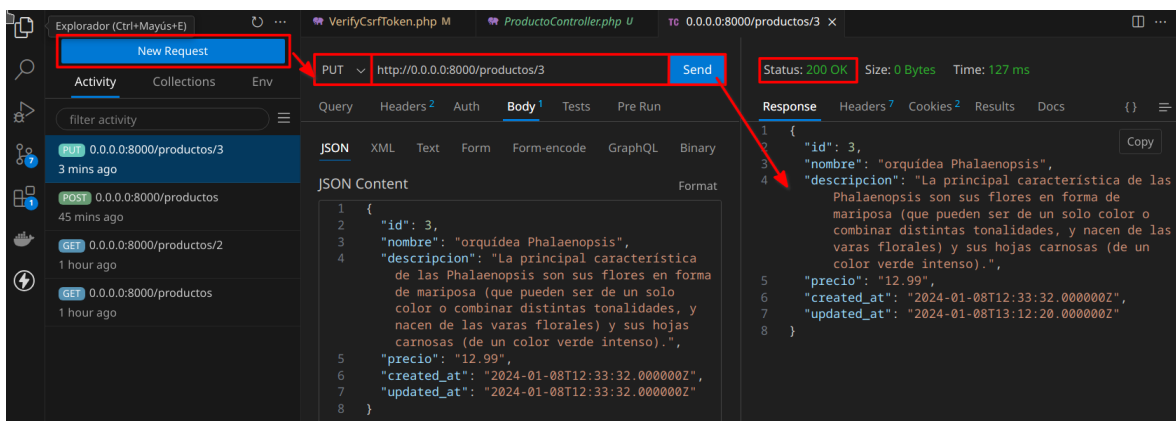
### 1.3.4. actualizar un producto existente

Recuerda añadir al fichero `VerifyCsrfToken.php` de la carpeta `app\Http\Middleware` la excepción:

```

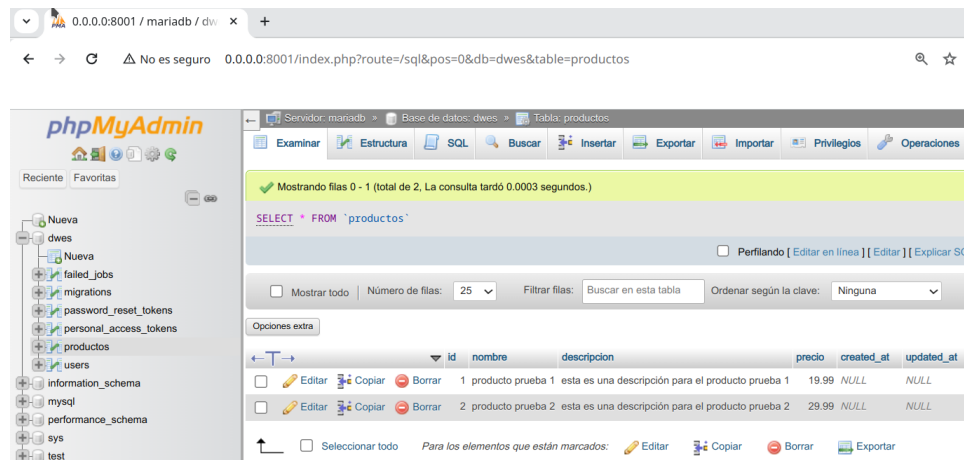
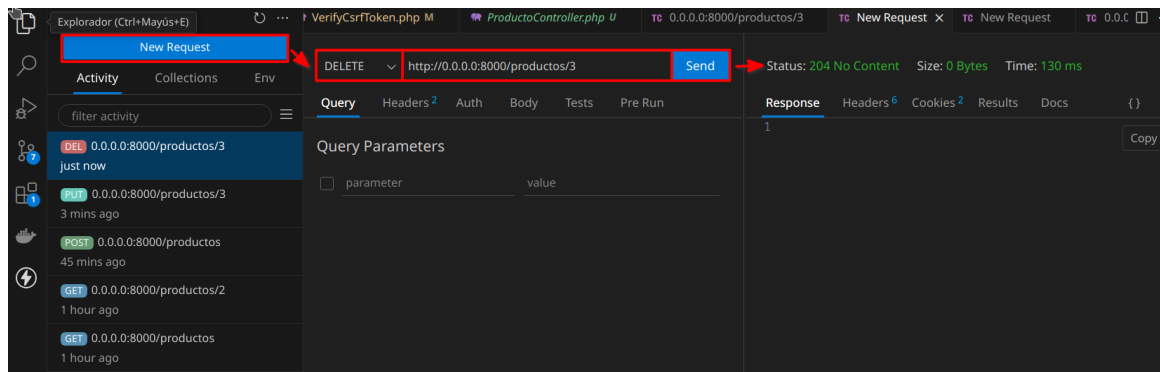
1 <?php
2 // [...]
3 protected $except = [
4     "http://0.0.0.0:8000/productos",
5     "http://0.0.0.0:8000/productos/3", // <-- esta nueva excepción
6 ];
7 }

```





### 1.3.5. eliminar un producto



## 2. ejercicios propuestos

### 2.1. Ejercicio 1

Sobre el proyecto **blog** de la sesión anterior, vamos a añadir estos cambios:

- Crea un controlador de tipo api llamado `PostController` en la carpeta `App\Http\Controllers\Api`, asociado al modelo `Post` que ya tenemos de sesiones previas. Rellena los métodos `index`, `show`, `store`, `update` y `destroy` para que, respectivamente, hagan lo siguiente:
  - `index` deberá devolver en formato JSON el listado de todos los posts, con un código 200
  - `show` deberá devolver la información del post que recibe, con un código 200
  - `store` deberá insertar un nuevo post con los datos recibidos, con un código 201, y utilizando el validador de posts que hiciste en la sesión 6. Para el usuario creador del post, pásale como parámetro JSON un usuario cualquiera de la base de datos.
  - `update` deberá modificar los campos del post recibidos, con un código 200, y empleando también el validador de posts que hiciste en la sesión 6.
  - `destroy` deberá eliminar el post recibido, devolviendo `null` con un código 204
- Crea una colección en *Thunder Client* llamada `Blog` que defina una petición para cada uno de los cinco servicios implementados. Comprueba que funcionan correctamente y exporta la colección a un archivo.

#### ¿Qué entregar?

Como entrega de esta sesión deberás comprimir el proyecto **blog** con los cambios incorporados, y eliminando las carpetas `vendor` y `node_modules` como se explicó en las sesiones anteriores. Añade dentro también la colección *Thunder Client* para probar los servicios. Renombra el archivo comprimido a `blog_08b.zip`.

## 3. bibliografia

---

- [Nacho Iborra Baeza](#).