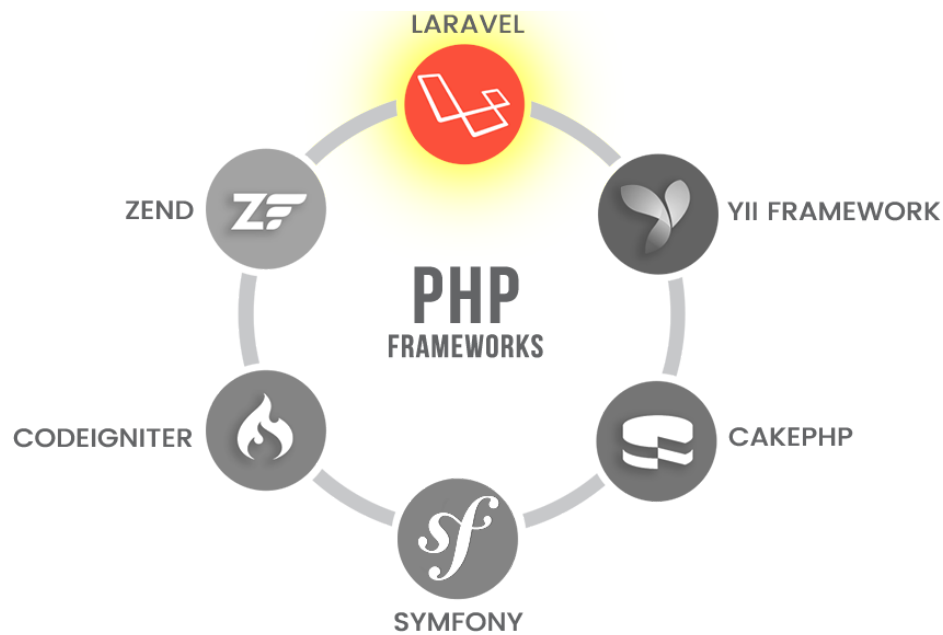


unidad didáctica 7

API RESTful en Laravel



1. **qué es API**
2. **qué es REST**
3. **ejemplo API Rest en tabla productos**
 3. 1. crear tabla productos
 3. 2. crear controlador ProductoController
 3. 3. cómo funciona la API REST
 3. 3. 1. listar todos los productos
 3. 3. 2. listar un producto en concreto
 3. 3. 3. introducir producto nuevo
 3. 3. 4. actualizar un producto existente
 3. 3. 5. eliminar un producto

1. qué es API

Una **API** (*Application Programming Interface*) es un conjunto de funciones y procedimientos por los cuales, una aplicación externa accede a los datos, a modo de biblioteca como una capa de abstracción y la API se encarga de enviar el dato solicitado.



Una de las características fundamentales de las API es que son **Sateless**, lo que quiere decir que las peticiones se hacen y desaparecen, no hay usuarios logueados ni datos que se quedan almacenados.

Ejemplos de APIs gratuitas:

- [ChuckNorris IO](#)
- [OMDB](#)
- [PokeAPI - Pokemon](#)
- [RAWg - Videojuegos](#)
- [The Star Wars API](#)

Para hacer pruebas con estas APIs podemos implementar el código para consumirlas o utilizar un cliente especial para el consumo de estos servicios.

- [PostMan](#)
- [Thunder Client](#) (utilizaremos esta extensión de VS Code para nuestras comprobaciones).
- [Insomnia](#)
- [Advance REST Client \(desde el navegador\)](#)

2. qué es REST

Con esta metodología llamada **REST** vamos a poder construir APIs para que desde un cliente externo se puedan consumir.

Gracias a este standard de la arquitectura del software vamos a poder montar un API que utilice los métodos standard `GET`, `POST`, `PUT` y `DELETE`.

3. ejemplo API Rest en tabla productos

3.1. crear tabla productos

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

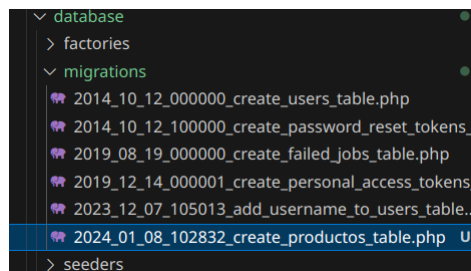
1. Crear **migración** para la tabla `productos`:

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/proyectos/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 }
```

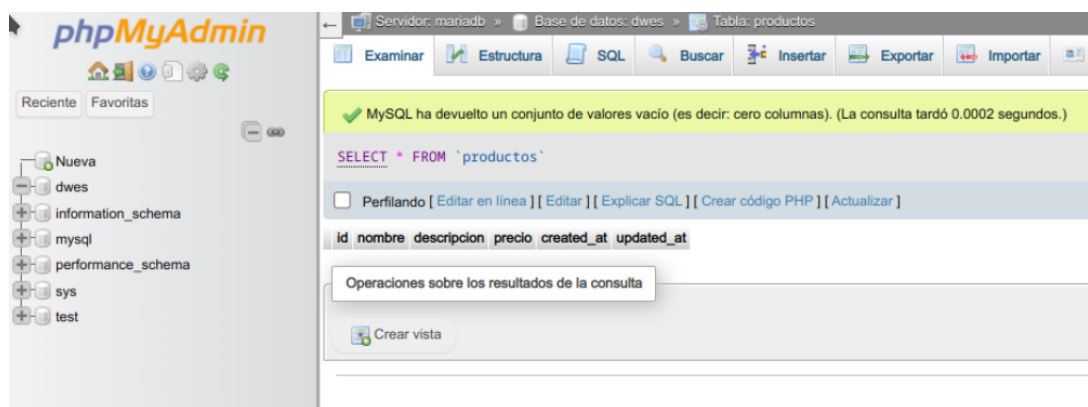
2. Ejecutar migración:

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

```
abc@jolly-wright:~/Escritorio/IES/DwES/proyectos/pru-udemy$ sudo docker-compose exec myapp php artisan migrate
```

INFO Running migrations.

2024_01_08_102832_create_productos_table 15ms DONE



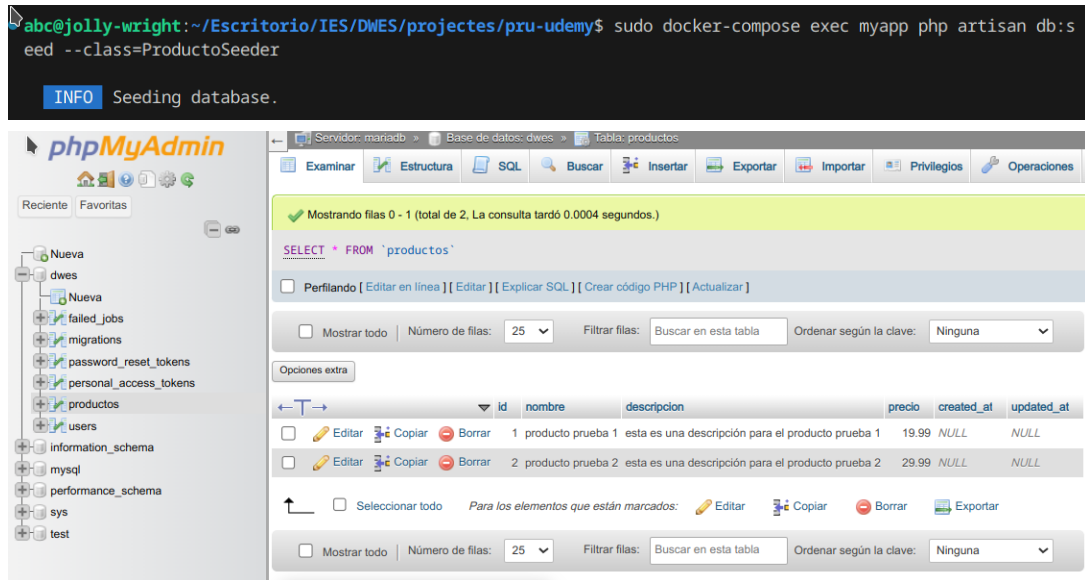
3. Crear un `seeder` para realizar una carga de datos:

Introducimos información en esta tabla nueva, creando un fichero en la carpeta `database/seeder` 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
```

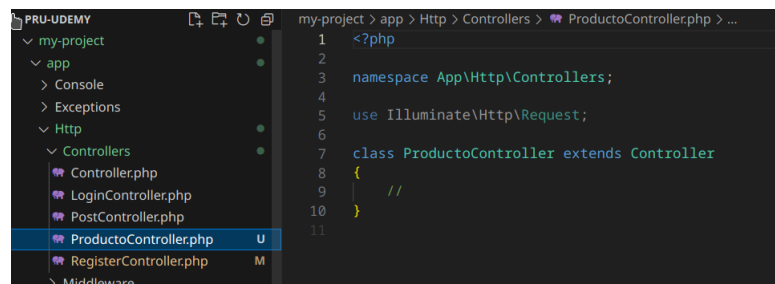
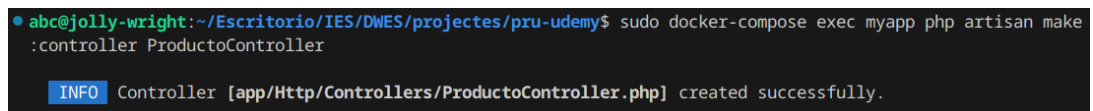


3.2. crear controlador ProductoController

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  }

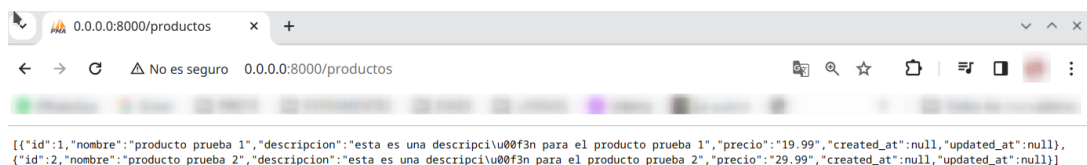
```

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 }

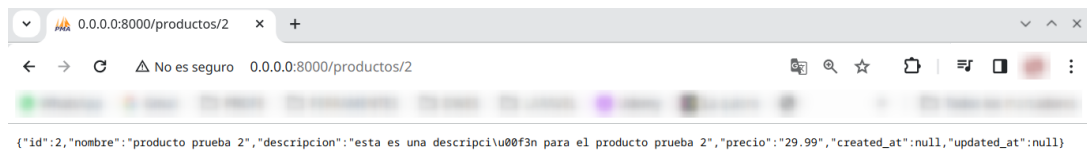
```

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`:

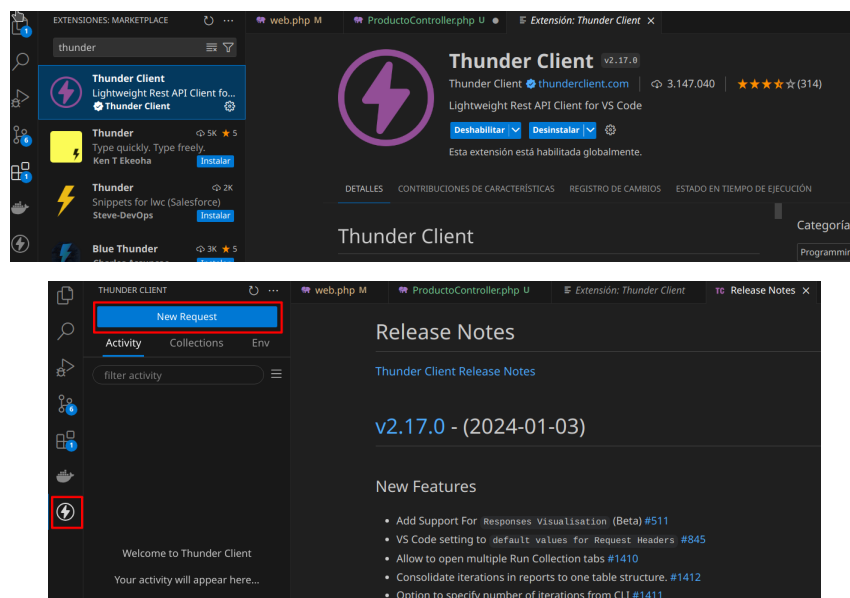
```
1     public function destroy($id){
2         Producto::findOrFail($id)->delete();
3
4         return response()->json(null, 204);
5     }
```

b) en `web.php`:

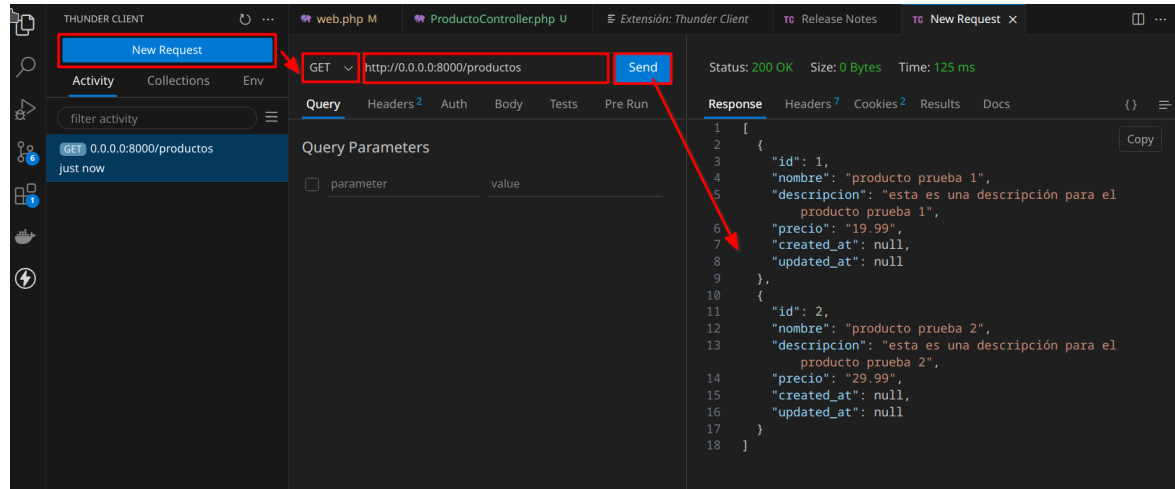
```
1 Route::prefix('productos')->group(function(){
2     Route::get('/',[ProductoController::class, 'index']);
3     Route::get('/{id}',[ProductoController::class, 'show']);
4     Route::post('/',[ProductoController::class, 'store']);
5     Route::put('/{id}',[ProductoController::class, 'update']);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

3.3. cómo funciona la API REST

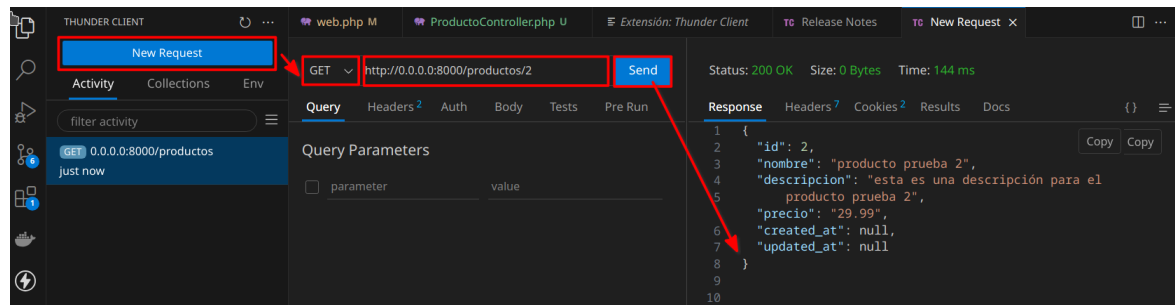
Para ello vamos a utilizar un software que es una extensión de Visual Studio Code, de nombre `Thunder Client`:



3.3.1. listar todos los productos



3.3.2. listar un producto en concreto



3.3.3. introducir producto nuevo

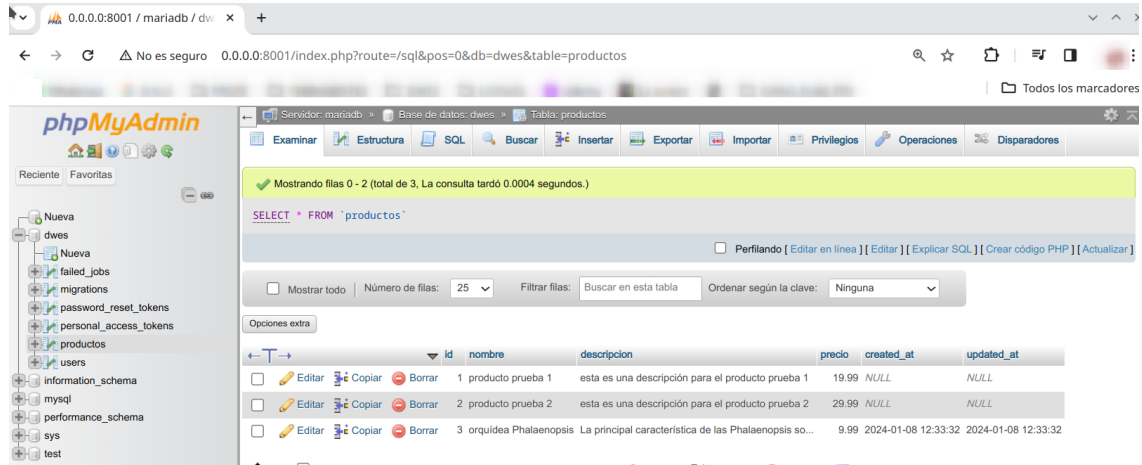
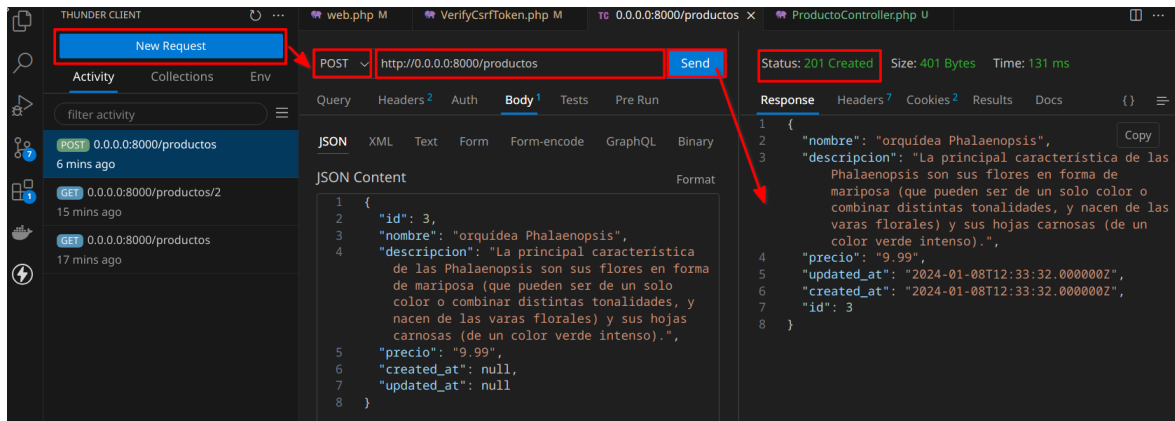
Si realizamos una nueva petición (new request) con método `post` y pasando (desde `body` y en `json`) un nuevo producto, va a mostrarnos un **error**.

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 }
```



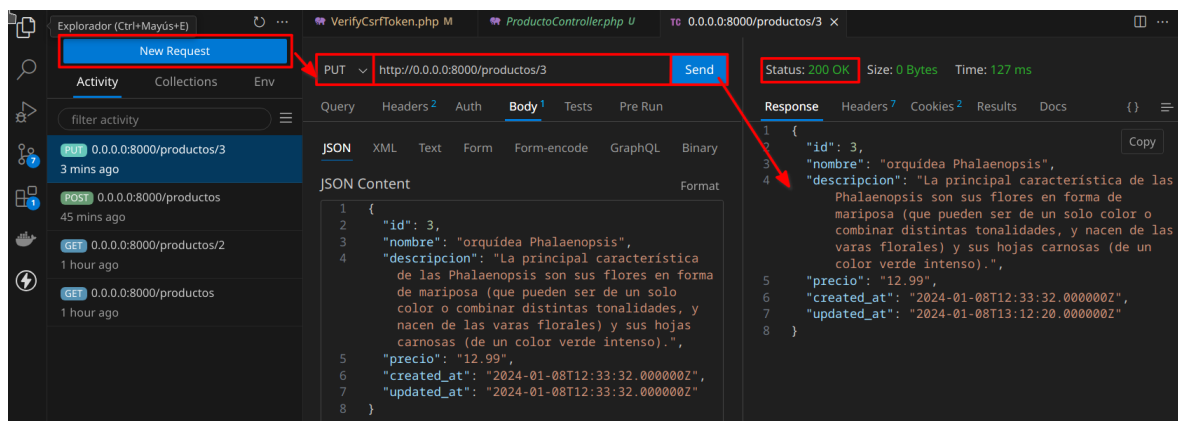
3.3.4. actualizar un producto existente

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 }

```





3.3.5. eliminar un producto

