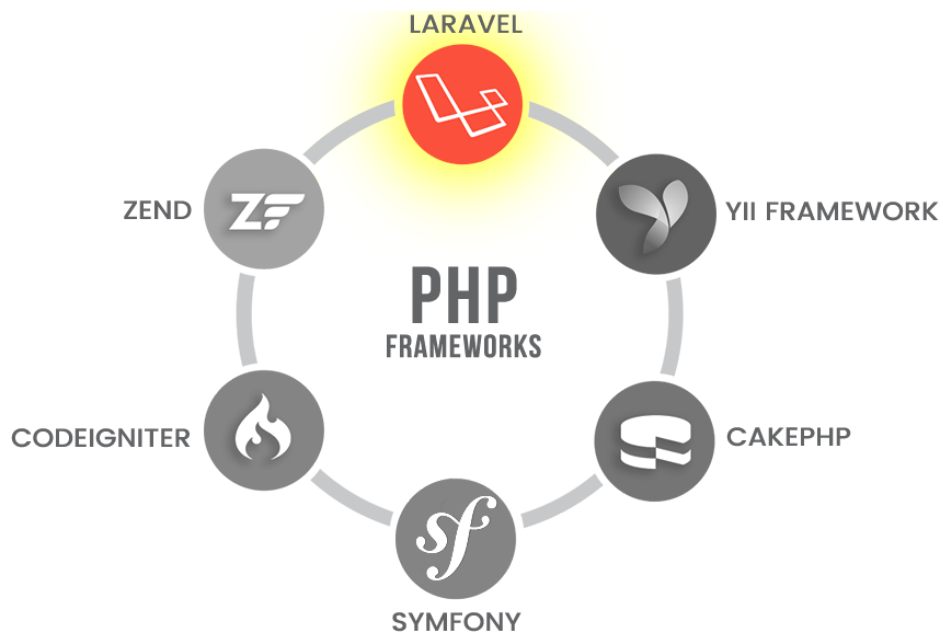


unidad didáctica 7

Laravel - servicios REST



1. conceptos de servicios REST

En esta unidad del curso veremos cómo emplear Laravel como proveedor de servicios REST. Comenzaremos detallando algunas cuestiones básicas de la arquitectura cliente-servidor y de los servicios REST, para luego pasar a ver cómo desarrollarlos y probarlos con Laravel.



A estas alturas todos deberíamos tener claro que cualquier aplicación web se basa en una arquitectura cliente-servidor, donde un servidor queda a la espera de conexiones de clientes, y los clientes se conectan a los servidores para solicitar ciertos recursos. Sobre esta base, veremos unas breves pinceladas de cómo funciona el protocolo HTTP, y en qué consisten los servicios REST.

1.1. Conceptos básicos del protocolo HTTP

Las comunicaciones web entre cliente y servidor se realizan mediante el protocolo **HTTP** (o HTTPS, en el caso de comunicaciones seguras). En ambos casos, cliente y servidor se envían cierta información estándar, en cada mensaje

En cuanto a los **clientes**, envían al servidor los datos del recurso que solicitan, junto con cierta información adicional, como por ejemplo las cabeceras de petición (información relativa al tipo de cliente o navegador, contenido que acepta, etc), y parámetros adicionales llamados normalmente *datos del formulario*, puesto que suelen contener la información de algún formulario que se envía de cliente a servidor.

Por lo que respecta a los **servidores**, aceptan estas peticiones, las procesan y envían de vuelta algunos datos relevantes, como un código de estado (indicando si la petición pudo ser atendida satisfactoriamente o no), cabeceras de respuesta (indicando el tipo de contenido enviado, tamaño, idioma, etc), y el recurso solicitado propiamente dicho, si todo ha ido correctamente.

Este es el mecanismo que hemos estado utilizando hasta ahora a través de los controladores: reciben la petición concreta del cliente, y envían una respuesta, que por el momento se ha centrado en renderizar un contenido HTML de una vista.

En cuanto a los **códigos de estado** de la respuesta, depende del resultado de la operación que se haya realizado, éstos se catalogan en cinco grupos:

- **Códigos 1xx:** representan información sobre una petición normalmente incompleta. No son muy habituales, pero se pueden emplear cuando la petición es muy larga, y se envía antes una cabecera para comprobar si se puede procesar dicha petición.
- **Códigos 2xx:** representan peticiones que se han podido atender satisfactoriamente. El código más habitual es el **200**, respuesta estándar para las peticiones que son correctas. Existen otras variantes, como el código **201**, que se envía cuando se ha insertado o creado un nuevo recurso en el servidor (una inserción en una base de datos, por ejemplo), o el código **204**, que indica que la petición se ha atendido bien, pero no se ha devuelto nada como respuesta.

- **Códigos 3xx:** son códigos de redirección, que indican que de algún modo la petición original se ha redirigido a otro recurso del servidor. Por ejemplo, el código *301* indica que el recurso solicitado se ha movido permanentemente a otra URL. El código *304* indica que el recurso solicitado no ha cambiado desde la última vez que se solicitó, por si se quiere recuperar de la caché local en ese caso.
- **Códigos 4xx:** indican un error por parte del cliente. El más típico es el error *404*, que indica que estamos solicitando una URL o recurso que no existe. Pero también hay otros habituales, como el *401* (cliente no autorizado), o *400* (los datos de la petición no son correctos, por ejemplo, porque los campos del formulario no sean válidos).
- **Códigos 5xx:** indican un error por parte del servidor. Por ejemplo, el error *500* indica un error interno del servidor, o el *504*, que es un error de *timeout* por tiempo excesivo en emitir la respuesta.

Haremos uso de estos códigos de estado en nuestros servicios REST para informar al cliente del tipo de error que se haya producido, o del estado en que se ha podido atender su petición.

1.2. Los servicios REST

REST son las siglas de **RE**presentational **St**ate **T**ransfer, y designa un estilo de arquitectura de aplicaciones distribuidas basado en HTTP. En un sistema REST, identificamos cada recurso a solicitar con una URI (identificador uniforme de recurso), y definimos un conjunto delimitado de comandos o métodos a realizar, que típicamente son:

- **GET:** para obtener resultados de algún tipo (listados completos o filtrados por alguna condición).
- **POST:** para realizar inserciones o añadir elementos en un conjunto de datos.
- **PUT:** para realizar modificaciones o actualizaciones del conjunto de datos.
- **DELETE:** para realizar borrados del conjunto de datos.
- Existen otros tipos de comandos o métodos, como por ejemplo **PATCH** (similar a PUT, pero para cambios parciales), **HEAD** (para consultar sólo el encabezado de la respuesta obtenida), etc.

Nos centraremos de momento en los cuatro métodos principales anteriores.

Por lo tanto, identificando el recurso a solicitar y el comando a aplicarle, el servidor que ofrece esta API REST proporciona una respuesta a esa petición. Esta respuesta típicamente viene dada por un mensaje en formato **JSON** o XML (aunque éste último cada vez está más en desuso). Esto permite que las aplicaciones puedan extenderse a distintas plataformas, y acceder a los mismos servicios desde una aplicación Angular, o una aplicación de escritorio .NET, o una aplicación móvil en Android, por poner varios ejemplos.

ACLARACIÓN: para quienes no conozcáis la definición de API (*Application Programming Interface*), básicamente es el conjunto de métodos o funcionalidades que se ponen a disposición de quienes los quieran utilizar. En este caso, el concepto de API REST hace referencia al conjunto de servicios REST proporcionados por el servidor para los clientes que quieran utilizarlos.

Una de las características fundamentales de las API es que son **Stateless**, 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*)

1.3. El formato JSON

JSON son las siglas de *JavaScript Object Notation*, una sintaxis propia de Javascript para poder representar objetos como cadenas de texto, y poder así serializar y enviar información de objetos a través de flujos de datos (archivos de texto, comunicaciones cliente-servidor, etc).

Un objeto Javascript se define mediante una serie de propiedades y valores. Por ejemplo, los datos de una persona (como nombre y edad) podríamos almacenarlos así:

```
1 let persona = {
2     nombre: "Nacho",
3     edad: 39
4 };
```

Este mismo objeto, convertido a JSON, formaría una cadena de texto con este contenido:

```
1 {"nombre": "Nacho", "edad": 39}
```

Del mismo modo, si tenemos una colección (vector) de objetos como ésta:

```
1 let personas = [
2     { nombre: "Nacho", edad: 39},
3     { nombre: "Mario", edad: 4},
4     { nombre: "Laura", edad: 2},
5     { nombre: "Nora", edad: 10}
6 ];
```

Transformada a JSON sigue la misma sintaxis, pero entre corchetes:

```
1 [{"nombre": "Nacho", "edad": 39}, {"nombre": "Mario", "edad": 4},
2 {"nombre": "Laura", "edad": 2}, {"nombre": "Nora", "edad": 10}]
```

Cuando solicitamos un servicio REST, típicamente la respuesta (los datos que nos envía el servicio) vienen en este formato JSON, de modo que es fácilmente serializable y se puede enviar entre cualquier tipo de cliente (móvil, web, escritorio) y el servidor.

2. creación de servicios REST

Veamos ahora qué pasos dar para construir una API REST en Laravel que dé soporte a las operaciones básicas sobre una o varias entidades: consultas (GET), inserciones (POST), modificaciones (PUT) y borrados (DELETE). Emplearemos para ello los denominados controladores de API, que comentamos brevemente en unidades anteriores, al hablar de controladores, y que proporcionan un conjunto de funciones ya definidas para dar soporte a cada uno de estos comandos.

2.1. 1. Definiendo los controladores de API

Para proporcionar una API REST a los clientes que lo requieran, necesitamos definir un controlador (o controladores) orientados a ofrecer estos servicios REST. Estos controladores en Laravel se denominan de tipo *api*, como vimos en sesiones previas. Normalmente se definirá un controlador API por cada uno de los modelos a los que necesitemos acceder. Vamos a crear uno de prueba para ofrecer una API REST sobre los libros de nuestra aplicación de biblioteca.

Existen diferentes formas de ejecutar el comando de creación del controlador de API. Aquí vamos a mostrar quizá una de las más útiles:

```
1 php artisan make:controller Api/LibroController --api --model=Libro
2 // ó, si no funciona:
3 // sudo docker-compose exec myapp php artisan make:controller
  Api/LibroController --api --model=Libro
```

```
1 namespace App\Http\Controllers\Api;
2
3 use App\Http\Controllers\Controller;
4 use App\Models\Libro;
5 use Illuminate\Http\Request;
6
7 class LibroController extends Controller
8 {
9     /**
10      * Display a listing of the resource.
11      *
12      * @return \Illuminate\Http\Response
13      */
14     public function index()
15     {
16         //
17     }
18
19     /**
20      * Store a newly created resource in storage.
21      *
22      * @param \Illuminate\Http\Request $request
23      * @return \Illuminate\Http\Response
24      */
25     public function store(Request $request)
```

```

26     {
27         //
28     }
29
30     /**
31      * Display the specified resource.
32      *
33      * @param  \App\Models\Libro  $libro
34      * @return \Illuminate\Http\Response
35      */
36     public function show(Libro $libro)
37     {
38         //
39     }
40
41     /**
42      * Update the specified resource in storage.
43      *
44      * @param  \Illuminate\Http\Request  $request
45      * @param  \App\Models\Libro  $libro
46      * @return \Illuminate\Http\Response
47      */
48     public function update(Request $request, Libro $libro)
49     {
50         //
51     }
52
53     /**
54      * Remove the specified resource from storage.
55      *
56      * @param  \App\Models\Libro  $libro
57      * @return \Illuminate\Http\Response
58      */
59     public function destroy(Libro $libro)
60     {
61         //
62     }
63 }

```

Observemos que se incorpora automáticamente la cláusula `use` para cargar el modelo asociado, que hemos indicado en el parámetro `--model`. Además, los métodos `show`, `update` y `destroy` ya vienen con un parámetro de tipo `Libro` que facilitará mucho algunas tareas.

NOTA: en el caso de versiones anteriores a Laravel 8, hay que tener en cuenta que, por defecto, los modelos se ubican en la carpeta `App`, por lo que deberemos indicar cualquier subcarpeta donde localizar el modelo cuando creamos el controlador, si es que lo hemos movido a una subcarpeta. Por ejemplo, `--model=Models/Libro`.

Cada una de las funciones del nuevo controlador creado se asocia a uno de los métodos REST comentados anteriormente:

- `index` se asociaría con una operación GET de listado general, para obtener todos los registros (de libros, en este caso)
- `store` se asociaría con una operación POST, para almacenar los datos que lleguen en la petición (como un nuevo libro, en nuestro caso)

- `show` se asociaría con una operación GET para obtener el registro asociado a un identificador concreto
- `update` se asociaría con una operación PUT, para actualizar los datos del registro asociado a un identificador concreto
- `destroy` se asociaría con una operación DELETE, para eliminar los datos del registro asociado a un identificador concreto

2.2. 2. Estableciendo las rutas

Una vez tenemos el controlador API creado, vamos a definir las rutas asociadas a cada método del controlador. Si recordamos de sesiones anteriores, podíamos emplear el método `Route::resource` en el archivo `routes/web.php` para establecer de golpe todas las rutas asociadas a un controlador de recursos. De forma análoga, podemos emplear el método `Route::apiResource` en el archivo `routes/api.php` para establecer automáticamente todas las rutas de un controlador de API. Añadimos esta línea en dicho archivo `routes/api.php`:

```
1 use App\Http\Controllers\Api\LibroController;
2 // ...
3 Route::apiResource('libros', LibroController::class);
```

Las rutas de API (aquellas definidas en el archivo `routes/api.php`) por defecto tienen un prefijo `api`, tal y como se establece en el *provider* `RouteServiceProvider`. Por tanto, hemos definido una ruta general `api/libros`, de forma que todas las subrutas que se deriven de ella llevarán a uno u otro método del controlador de API de libros.

Podemos comprobar **qué rutas hay activas** con este comando:

```
1 php artisan route:list
2 // ó, si no funciona:
3 // sudo docker-compose exec myapp php artisan route:list
```

Veremos, entre otras, las 5 rutas derivadas del controlador API de libros:

```
1 +-----+
2 | Method      | URI                | Name          |
3 +-----+
4 | GET|HEAD    | api/libros         | libros.index  |
5 | POST        | api/libros         | libros.store  |
6 | GET|HEAD    | api/libros/{libro} | libros.show   |
7 | PUT|PATCH  | api/libros/{libro} | libros.update |
8 | DELETE     | api/libros/{libro} | libros.destroy|
9 +-----+
```

2.3. 3. Servicios GET

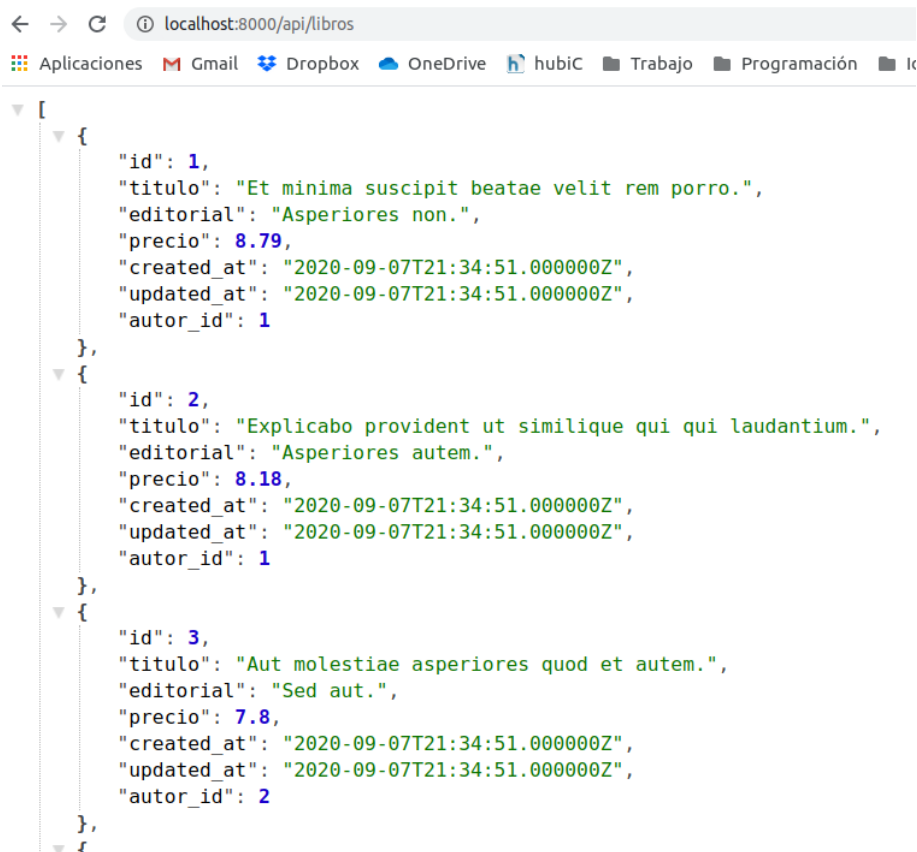
Vamos a empezar por definir el método `index`. En este caso, vamos a obtener el conjunto de libros de la base de datos y devolverlo tal cual:

```

1 public function index()
2 {
3     $libros = Libro::get();
4     return $libros;
5 }

```

Si accedemos a la ruta `api/libros` desde el navegador, se activará el método `index` que acabamos de implementar, y recibiremos los libros de la base de datos, directamente en formato JSON.



```

[
  {
    "id": 1,
    "titulo": "Et minima suscipit beatae velit rem porro.",
    "editorial": "Asperiores non.",
    "precio": 8.79,
    "created_at": "2020-09-07T21:34:51.000000Z",
    "updated_at": "2020-09-07T21:34:51.000000Z",
    "autor_id": 1
  },
  {
    "id": 2,
    "titulo": "Explicabo provident ut similique qui qui laudantium.",
    "editorial": "Asperiores autem.",
    "precio": 8.18,
    "created_at": "2020-09-07T21:34:51.000000Z",
    "updated_at": "2020-09-07T21:34:51.000000Z",
    "autor_id": 1
  },
  {
    "id": 3,
    "titulo": "Aut molestiae asperiores quod et autem.",
    "editorial": "Sed aut.",
    "precio": 7.8,
    "created_at": "2020-09-07T21:34:51.000000Z",
    "updated_at": "2020-09-07T21:34:51.000000Z",
    "autor_id": 2
  }
]

```

NOTA: podemos instalar la extensión [JSON formatter](#) para Chrome, y así poder ver los datos en formato JSON más organizados y con la sintaxis resaltada, como en la imagen anterior.

De una forma similar, podríamos implementar y probar el método `show`, para mostrar los datos de un libro en particular:

```

1 public function show(Libro $libro)
2 {
3     return $libro;
4 }

```

En este caso, si accedemos a la URI `api/libros/1`, obtendremos la información del libro con `id = 1`. Notar que Laravel se encarga automáticamente de buscar el libro por nosotros (hacer la correspondiente operación `find` para el `id` proporcionado). Es lo que se conoce como *enlace implícito*, y es algo que también está disponible en los controladores web normales, siempre que los asociemos correctamente con el modelo vinculado. Esto se hace automáticamente si creamos

el controlador junto con el modelo, como vimos en la unidad 4, o si usamos el parámetro `--model` para asociarlo, como hemos hecho aquí.

2.3.1. 3.1. Más sobre el formato JSON y la respuesta

Tras probar los dos servicios anteriores, habrás observado que Laravel se encarga de transformar directamente los registros obtenidos a formato JSON cuando los enviamos mediante `return`, por lo que, en principio, no tenemos por qué preocuparnos de este proceso. Sin embargo, de este modo se escapan algunas cosas a nuestro control. Por ejemplo, y sobre todo, no podemos especificar el código de estado de la respuesta, que por defecto es 200 si todo ha ido correctamente. Además, tampoco podemos controlar qué información enviar del objeto en cuestión.

Si queremos limitar o formatear la información a enviar de los objetos que estamos tratando, y que no se envíen todos sus campos sin más, tenemos varias opciones:

- Añadir cláusulas `hidden` en los modelos correspondientes, para indicar que esa información no debe ser enviada en ningún caso en ninguna parte de la aplicación. Es lo que ocurre, por ejemplo, con el campo *password* del modelo de `Usuario`:

```
1 protected $hidden = ['password'];
```

- Definir a mano un array con los campos a enviar en el método del controlador. En el caso de la ficha del libro anterior, si sólo queremos enviar el título y la editorial, podríamos hacer algo así:

```
1 public function show(Libro $libro)
2 {
3     return [
4         'titulo'    => $libro->titulo,
5         'editorial' => $libro->editorial
6     ];
7 }
```

- En el caso de que el paso anterior sea muy costoso (porque el modelo tenga muchos campos, o porque tengamos que hacer lo mismo en varias partes del código), también podemos definir recursos (*resources*), que permiten separar el código de la información a mostrar del propio controlador. [Aquí](#) podéis encontrar información al respecto, ya que estos contenidos escapan del alcance de esta sesión.

Por otra parte, si queremos añadir o modificar más información en la respuesta, como el código de estado, la estructura anterior no nos sirve, ya que siempre se va a enviar un código 200. Para esto, es conveniente emplear el método `response()->json(...)`, que permite especificar como primer parámetro los datos a enviar, y como segundo parámetro el código de estado. Los métodos anteriores quedarían así, enviando un código 200 como respuesta (aunque si se omite el segundo parámetro, se asume que es 200):

```

1 public function index()
2 {
3     $libros = Libro::get();
4     return response()->json($libros, 200);
5 }
6 // ...
7 public function show(Libro $libro)
8 {
9     return response()->json($libro, 200);
10 }

```

2.4. 4. Resto de servicios

Veamos ahora cómo implementar el resto de servicios (POST, PUT y DELETE). En el caso de la inserción (**POST**), deberemos recibir en la petición los datos del objeto a insertar (un libro, en nuestro ejemplo). Igual que los datos del servidor al cliente se envían en formato JSON, es de esperar en aplicaciones que siguen la arquitectura REST que los datos del cliente al servidor también se envíen en formato JSON.

Nuestro método `store`, asociado al servicio POST, podría quedar de este modo (devolvemos el código de estado 201, que se utiliza cuando se han insertado elementos nuevos):

```

1 public function store(Request $request)
2 {
3     $libro = new Libro();
4     $libro->titulo = $request->titulo;
5     $libro->editorial = $request->editorial;
6     $libro->precio = $request->precio;
7     $libro->autor()->associate(Autor::findOrFail($request->autor_id));
8     $libro->save();
9
10    return response()->json($libro, 201);
11 }

```

De forma similar implementaríamos el servicio **PUT**, a través del método `update`. En este caso devolvemos un código de estado 200:

```

1 public function update(Request $request, Libro $libro)
2 {
3     $libro->titulo = $request->titulo;
4     $libro->editorial = $request->editorial;
5     $libro->precio = $request->precio;
6     $libro->autor()->associate(Autor::findOrFail($request->autor_id));
7     $libro->save();
8
9     return response()->json($libro);
10 }

```

Finalmente, para el servicio **DELETE**, debemos implementar el método `destroy`, que podría quedar así:

```

1 public function destroy(Libro $libro)
2 {
3     $libro->delete();
4     return response()->json(null, 204);
5 }

```

Notar que devolvemos un código de estado 204, que indica que no estamos devolviendo contenido (es *null*). Por otra parte, es habitual en este tipo de operaciones de borrado devolver en formato JSON el objeto que se ha eliminado, por si acaso se quiere deshacer la operación en un paso posterior. En este caso, el código del método de borrado sería así:

```

1 public function destroy(Libro $libro)
2 {
3     $libro->delete();
4     return response()->json($libro);
5 }

```

Como podemos empezar a intuir, probar estos servicios no es tan sencillo como probar servicios de tipo GET, ya que no podemos simplemente teclear una URL en el navegador. Necesitamos un mecanismo para pasarle los datos al servidor en formato JSON, y también el método (POST, PUT o DELETE). Veremos cómo en la siguiente sección.

2.4.1. 4.1. Validación de datos

A la hora de recibir datos en formato JSON para servicios REST, también podemos establecer mecanismos de **validación** similares a los vistos para los formularios, a través de los correspondientes *requests*. De hecho, en el caso de la biblioteca podemos emplear la clase `App\Http\Requests\LibroPost` que hicimos en sesiones anteriores, para validar que los datos que llegan tanto a `store` como a `update` son correctos. Basta con usar un parámetro de este tipo en estos métodos, en lugar del parámetro `Request` que viene por defecto:

```

1 public function store(LibroPost $request)
2 {
3     // ...
4 }
5 ...
6 public function update(LibroPost $request, Libro $libro)
7 {
8     // ...
9 }

```

2.4.2. 4.2. Respuestas de error

Por otra parte, debemos asegurarnos de que cualquier error que se produzca en la parte de la API devuelva un contenido en formato JSON, y no una página web. Por ejemplo, si solicitamos ver la ficha de un libro cuyo *id* no existe, no debería devolvernos una página de error 404, sino un código de estado 404 con un mensaje de error en formato JSON.

Esto no se cumple por defecto, ya que Laravel está configurado para renderizar una vista con el error producido. Para modificar este comportamiento en **versiones anteriores a Laravel 8**, debemos editar el archivo `App\Exceptions\Handler.php`, en concreto su método `render`, y hacer algo así:

```

1 public function render($request, Throwable $exception)
2 {
3     if ($request->is('api*'))
4     {
5         if ($exception instanceof ModelNotFoundException)
6             return response()->json(['error' => 'Elemento no encontrado'], 404);
7         else if ($exception instanceof ValidationException)
8             return response()->json(['error' => 'Datos no válidos'], 400);
9         else if (isset($exception))
10            return response()->json(['error' => 'Error en la aplicación: ' .
11                                   $exception->getMessage()], 500);
12    }
13
14    // Esta es la única instrucción que hay en la versión original
15    return parent::render($request, $exception);
16 }

```

Hemos añadido sobre el código original una cláusula `if` que se centra en las peticiones de tipo `api`. En este caso, podemos distinguir los distintos tipos de excepciones que se producen. Para nuestro ejemplo distinguimos tres: errores de tipo 404, errores de validación u otros errores. En todos los casos se devuelve un contenido JSON con el código de estado y campos adecuados. Si todo es correcto y no hay errores, o si no estamos en rutas `api`, el comportamiento será el habitual.

En el caso de **Laravel 8 y posteriores**, el método a modificar se llama `register`, dentro de la misma clase `App\Exceptions\Handler.php`. Lo podemos dejar de este modo para hacer algo equivalente a lo anterior:

```

1 public function register()
2 {
3     $this->renderable(function (Throwable $exception) {
4         if (request()->is('api*'))
5         {
6             if ($exception instanceof ModelNotFoundException)
7                 return response()->json(['error' => 'Recurso no encontrado'],
8                                         404);
9             else if ($exception instanceof ValidationException)
10                return response()->json(['error' => 'Datos no válidos'], 400);
11            else if (isset($exception))
12                return response()->json(['error' => 'Error: ' .
13                                       $exception->getMessage()], 500);
14        }
15    });
16 }

```

NOTA: relacionado con el código anterior, las excepciones que se identifican están en `Illuminate\Database\Eloquent\ModelNotFoundException` e `Illuminate\Validation\ValidationException`, respectivamente.

En algunas versiones de Laravel, la clase base `ExceptionHandler` realiza algunas conversiones de tipos de excepciones. Así, por ejemplo, la excepción `ModelNotFoundException` se convierte a `NotFoundHttpException`. En este caso, el `if` anterior que detecta la excepción podría no funcionar, ya que el operador `instanceof` está buscando la excepción equivocada. Una forma

algo más completa de detectar si no se encuentra el recurso solicitado sería esta (incluimos las cláusulas `using` correspondientes también):

```

1 namespace App\Exceptions;
2
3 use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;
4 use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
5 use Illuminate\Validation\ValidationException;
6 use Illuminate\Database\Eloquent\ModelNotFoundException;
7 use Throwable;
8
9 class Handler extends ExceptionHandler
10 {
11     // ...
12
13     public function register()
14     {
15         $this->renderable(function (Throwable $exception) {
16             if (request()->is('api*'))
17             {
18                 if ($exception instanceof ModelNotFoundException ||
19                     ($exception instanceof NotFoundHttpException &&
20                     $exception->getPrevious() &&
21                     $exception->getPrevious() instanceof ModelNotFoundException))
22                     return response()->json(['error' => 'Recurso no encontrado'],
23                     404);
24                 else if ($exception instanceof ValidationException)
25                     return response()->json(['error' => 'Datos no válidos'], 400);
26                 else if (isset($exception))
27                     return response()->json(['error' => 'Error: ' .
28                     $exception->getMessage()], 500);
29             }
30         });
31     }
32 }
```

De este modo, detectamos tanto si es una `ModelNotFoundException` original como si ha sido convertida a `NotFoundHttpException`.

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:

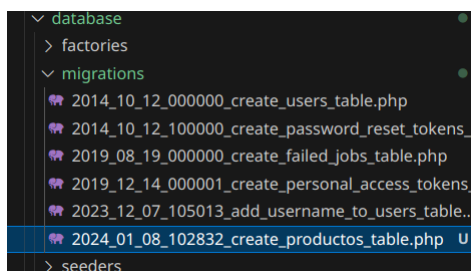
1. Crear **migración** para la tabla `productos`:

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/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.
```

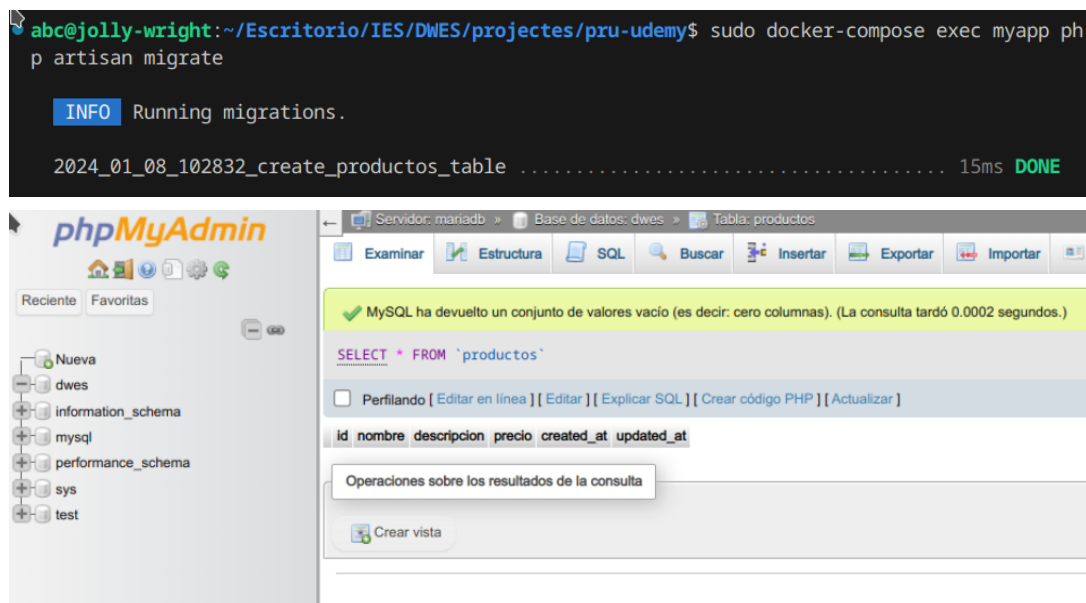


2. Añadir al fichero generado (en la carpeta `migrations` y en el ejemplo anterior `2024_01_08_102832_create_productos_table.php`) el resto de campos que se requieran en la tabla `productos`:

```
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     }

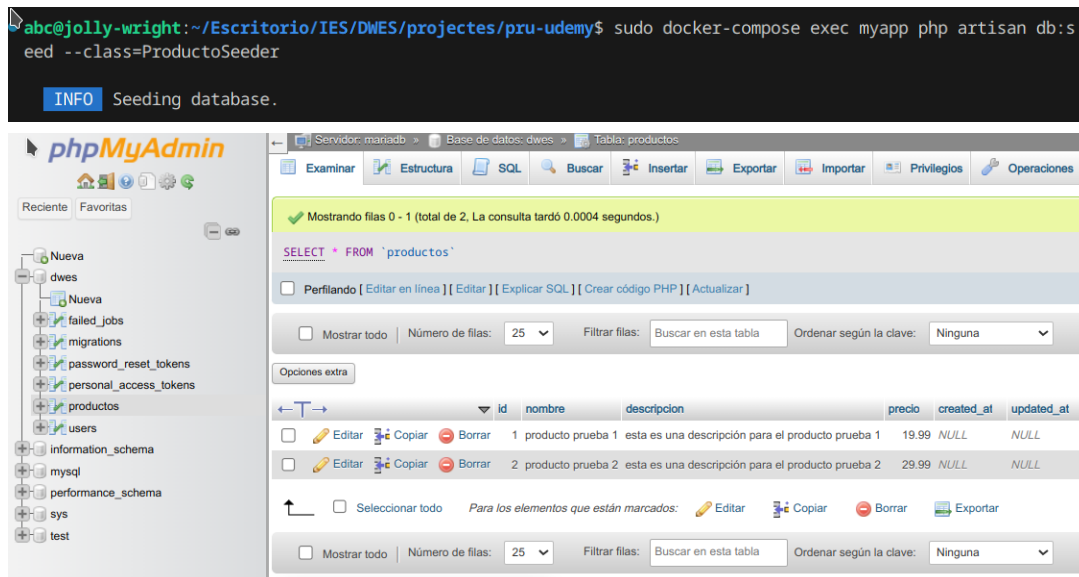
```

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

```

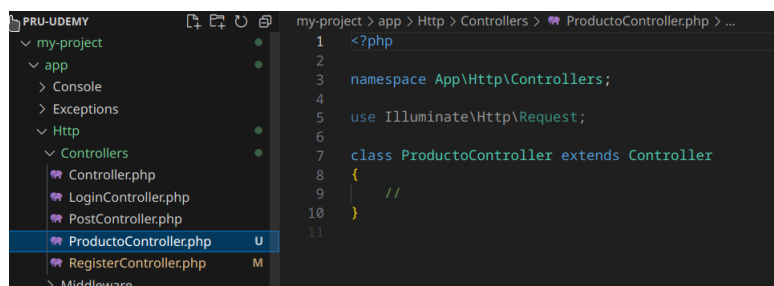
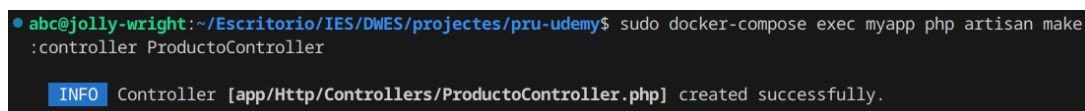


3.2. crear controlador ProductoController

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  }

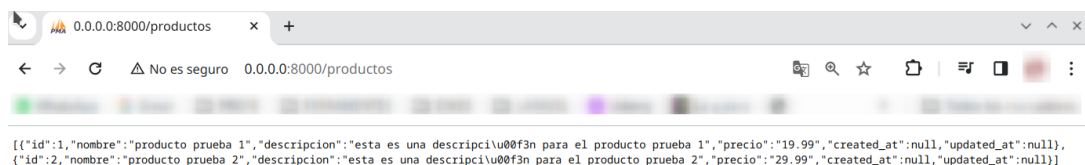
```

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 }

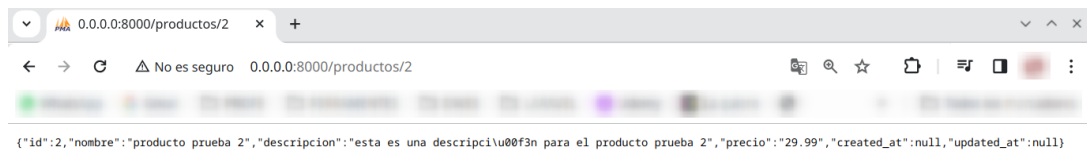
```

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`:

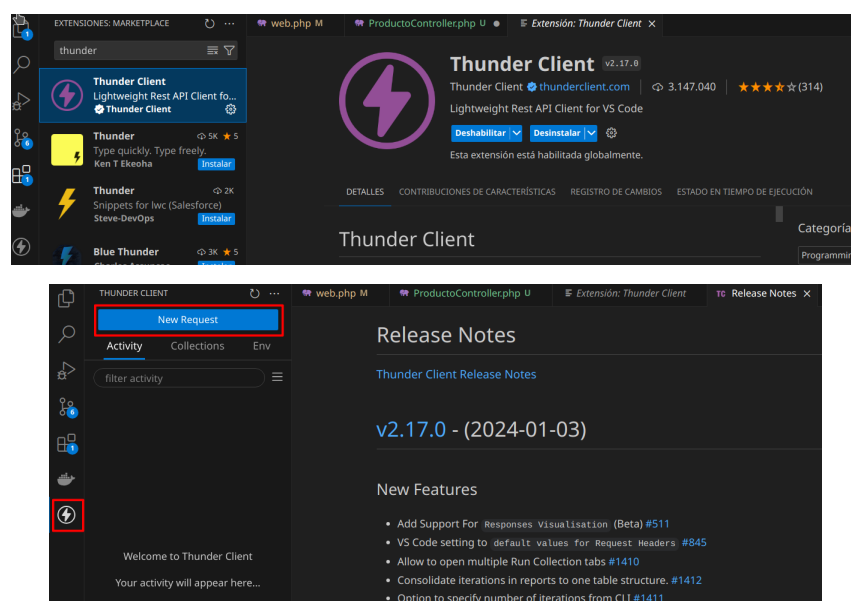
```
1     public function destroy($id){
2         Producto::findOrFail($id)->delete();
3
4         return response()->json(null, 204);
5     }
```

b) en `web.php`:

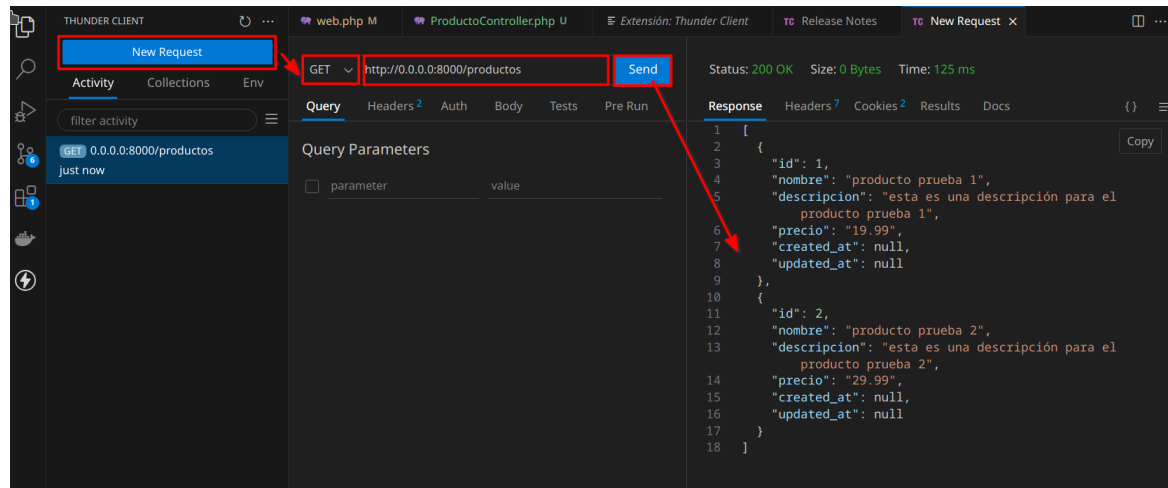
```
1 Route::prefix('productos')->group(function(){
2     Route::get('/',[ProductoController::class, 'index']);
3     Route::get('/{id}',[ProductoController::class, 'show']);
4     Route::post('/',[ProductoController::class, 'store']);
5     Route::put('/{id}',[ProductoController::class, 'update']);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

3.3. cómo funciona la API REST

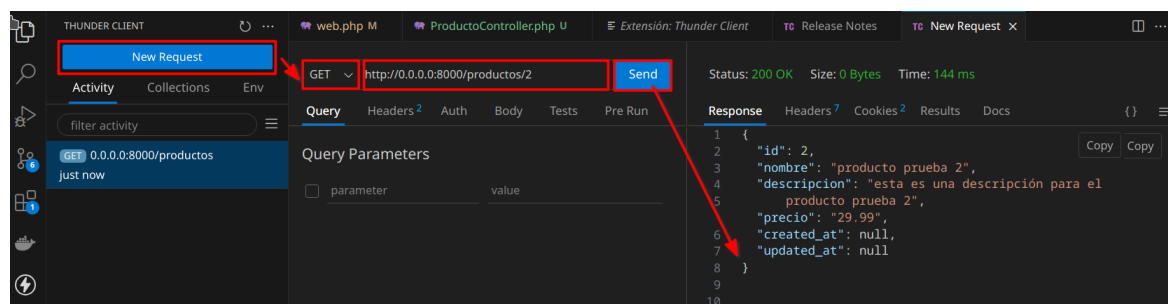
Para ello vamos a utilizar un software que es una extensión de Visual Studio Code, de nombre `Thunder Client`:



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`.

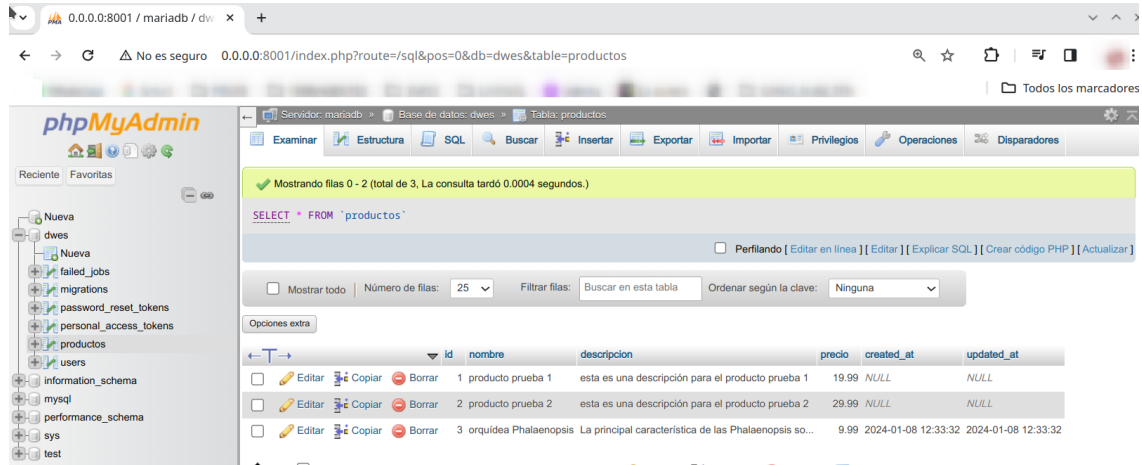
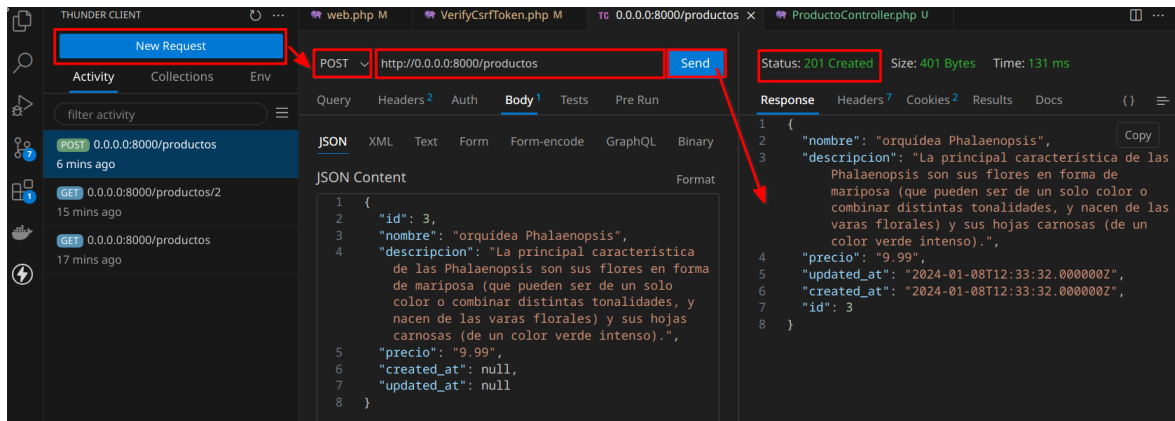
Ya que, ahora mismo, estamos realizando pruebas, vamos a indicarle a Laravel que excluya la URL en cuestión de la verificación.

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  }

```



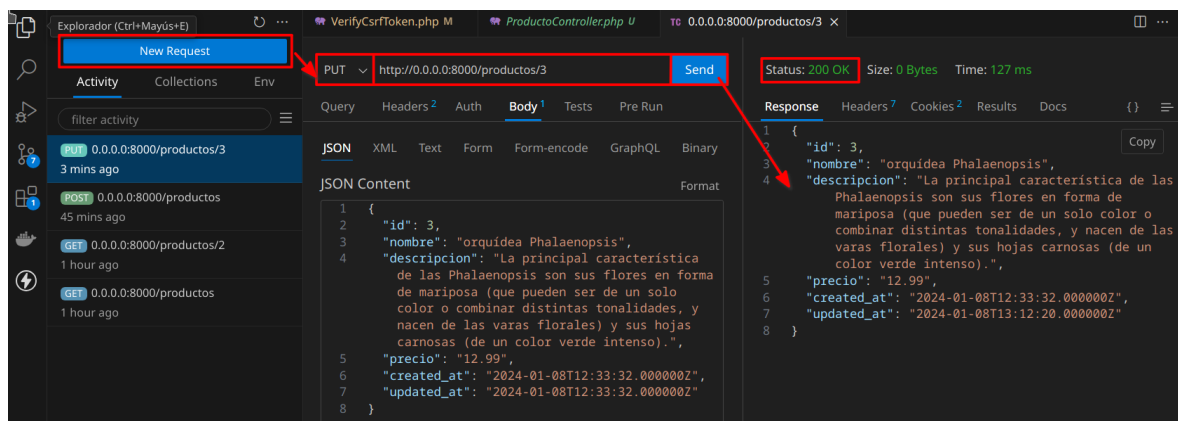
3.3.4. actualizar un producto existente

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

```

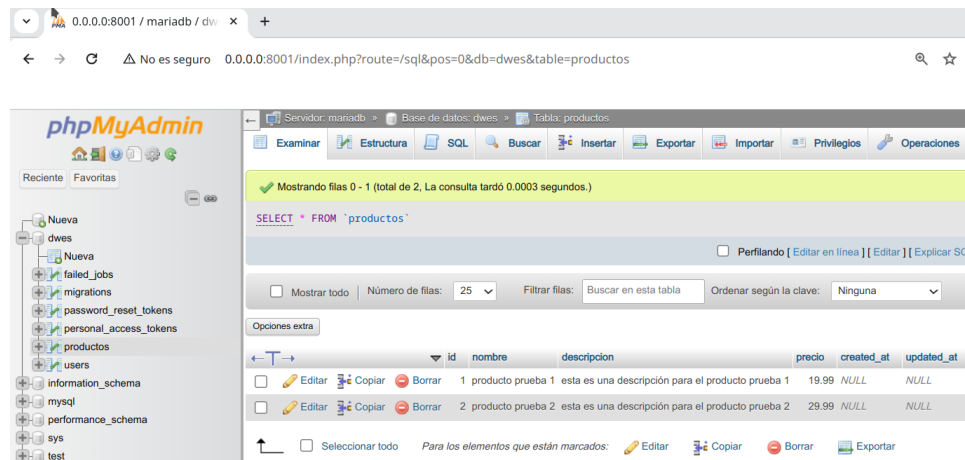
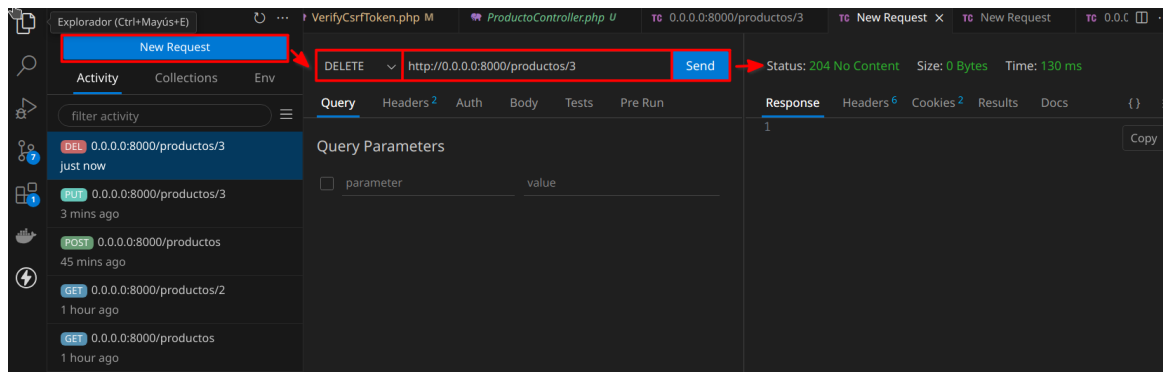
1 <?php
2 // [...]
3 protected $except = [
4     "http://0.0.0.0:8000/productos",
5     "http://0.0.0.0:8000/productos/3", // <-- esta nueva excepción
6 ];
7 }

```





3.3.5. eliminar un producto



4. ejercicios propuestos

4.1. 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_08.zip`.

5. bibliografia

- [Nacho Iborra Baeza](#).