

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.

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:

```
php artisan make:controller Api/LibroController --api --model=Libro
```

Esto creará el controlador en la carpeta `App\Http\Controllers\Api` con una serie de funciones ya predefinidas. No es obligatorio ubicarlo en esa subcarpeta, obviamente, pero esto nos servirá para separar los controladores de API del resto. Esta será la apariencia del controlador generado:

```
namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use App\Models\Libro;
use Illuminate\Http\Request;

class LibroController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        //
    }

    /**
     * Display the specified resource.
     *
     * @param \App\Models\Libro $libro
     * @return \Illuminate\Http\Response
     */
    public function show(Libro $libro)
    {
        //
    }

    /**
     * Update the specified resource in storage.
     *
     * @param \Illuminate\Http\Request $request
     * @param \App\Models\Libro $libro
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, Libro $libro)
    {

```

```
        //
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param \App\Models\Libro $libro
     * @return \Illuminate\Http\Response
     */
    public function destroy(Libro $libro)
    {
        //
    }
}
```

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. 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`:

```
use App\Http\Controllers\Api\LibroController;
...
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:

```
php artisan route:list
```

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

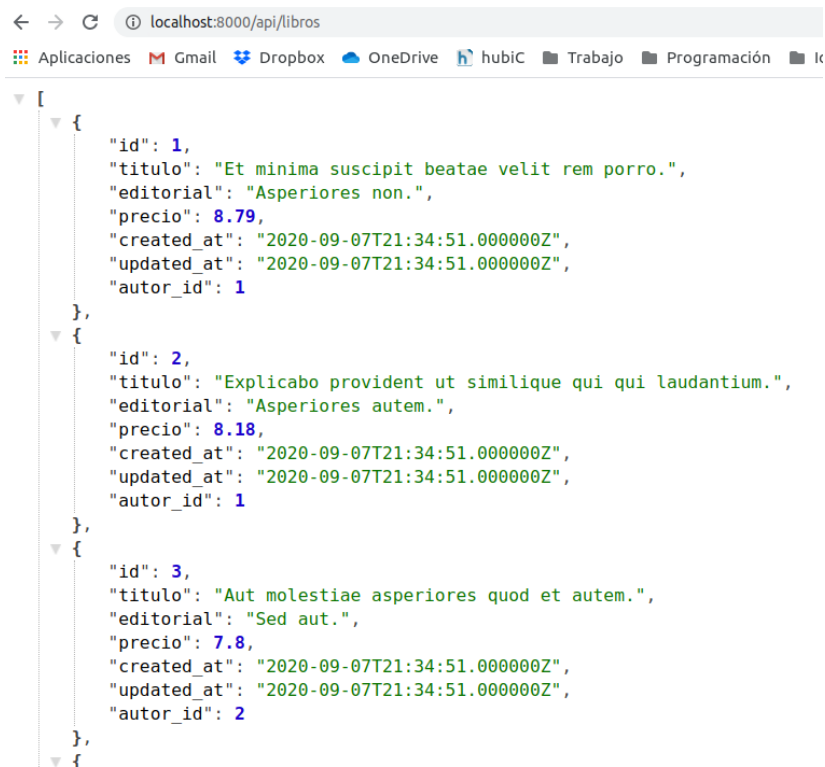
```
+-----+
|Method   | URI                               | Name           |
+-----+
| GET|HEAD | api/libros                        | libros.index   |
| POST    | api/libros                        | libros.store   |
| GET|HEAD | api/libros/{libro}               | libros.show    |
| PUT|PATCH | api/libros/{libro}              | libros.update  |
| DELETE  | api/libros/{libro}              | libros.destroy |
+-----+
```

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:

```
public function index()
{
    $libros = Libro::get();
    return $libros;
}
```

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:

```

public function show(Libro $libro)
{
    return $libro;
}

```

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.

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`:

```
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:

```
public function show(Libro $libro)
{
    return [
        'titulo' => $libro->titulo,
        'editorial' => $libro->editorial
    ];
}
```

- 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):

```
public function index()
{
    $libros = Libro::get();
    return response()->json($libros, 200);;
}
...
public function show(Libro $libro)
{
    return response()->json($libro, 200);
}
```

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):

```
public function store(Request $request)
{
    $libro = new Libro();
    $libro->titulo = $request->titulo;
    $libro->editorial = $request->editorial;
    $libro->precio = $request->precio;
    $libro->autor()->associate(Autor::findOrFail($request->autor_id));
    $libro->save();

    return response()->json($libro, 201);
}
```

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

```
public function update(Request $request, Libro $libro)
{
    $libro->titulo = $request->titulo;
    $libro->editorial = $request->editorial;
    $libro->precio = $request->precio;
    $libro->autor()->associate(Autor::findOrFail($request->autor_id));
    $libro->save();

    return response()->json($libro);
}
```

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

```
public function destroy(Libro $libro)
{
    $libro->delete();
    return response()->json(null, 204);
}
```

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:

```
public function destroy(Libro $libro)
{
    $libro->delete();
    return response()->json($libro);
}
```

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.

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:

```
public function store(LibroPost $request)
{
    ...
}
...
public function update(LibroPost $request, Libro $libro)
{
    ...
}
```

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:

```
public function render($request, Throwable $exception)
{
    if ($request->is('api*'))
    {
        if ($exception instanceof ModelNotFoundException)
            return response()->json(['error' => 'Elemento no encontrado'],
                404);
        else if ($exception instanceof ValidationException)
            return response()->json(['error' => 'Datos no válidos'], 400);
        else if (isset($exception))
            return response()->json(['error' => 'Error en la aplicación: ' .
                $exception->getMessage()], 500);
    }

    // Esta es la única instrucción que hay en la versión original
    return parent::render($request, $exception);
}
```

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:

```
public function register()
{
    $this->renderable(function (Throwable $exception) {
        if (request()->is('api*'))
        {
            if ($exception instanceof ModelNotFoundException)
                return response()->json(['error' => 'Recurso no encontrado'],
                    404);
            else if ($exception instanceof ValidationException)
                return response()->json(['error' => 'Datos no válidos'],
                    400);
            else if (isset($exception))
                return response()->json(['error' => 'Error: ' .
                    $exception->getMessage()], 500);
        }
    });
}
```

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):

```
namespace App\Exceptions;

use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
use Illuminate\Validation\ValidationException;
use Illuminate\Database\Eloquent\ModelNotFoundException;
use Throwable;

class Handler extends ExceptionHandler
{
    ...

    public function register()
    {
        $this->renderable(function (Throwable $exception) {
            if (request()->is('api*'))
            {
                if ($exception instanceof ModelNotFoundException ||
                    ($exception instanceof NotFoundHttpException &&
                    $exception->getPrevious() &&
                    $exception->getPrevious() instanceof ModelNotFoundException))
                    return response()->json(['error' => 'Recurso no encontrado'],
                        404);
                else if ($exception instanceof ValidationException)
                    return response()->json(['error' => 'Datos no válidos'],
                        400);
                else if (isset($exception))
                    return response()->json(['error' => 'Error: ' .
                    $exception->getMessage()], 500);
            }
        });
    }
}
```

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