



Sesión 8. Desarrollo de servicios REST

En esta última sesión 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.

1. Introducción a los servicios REST

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. El 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 *REpresentational State Transfer*, y designa un estilo de arquitectura de aplicaciones distribuidas basada 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 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.

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:

```
let persona = {  
  nombre: "Nacho",  
  edad: 39  
};
```

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

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

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

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

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

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

2. Construyendo una API REST básica

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 la sesión 3, y que proporcionan un conjunto de funciones ya definidas para dar soporte a cada uno de estos comandos.

2.1. 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 necesitamos 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

2.2. 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`:

```
// Laravel < 8
Route::apiResource('libros', Api\LibroController::class);

// Laravel 8
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:

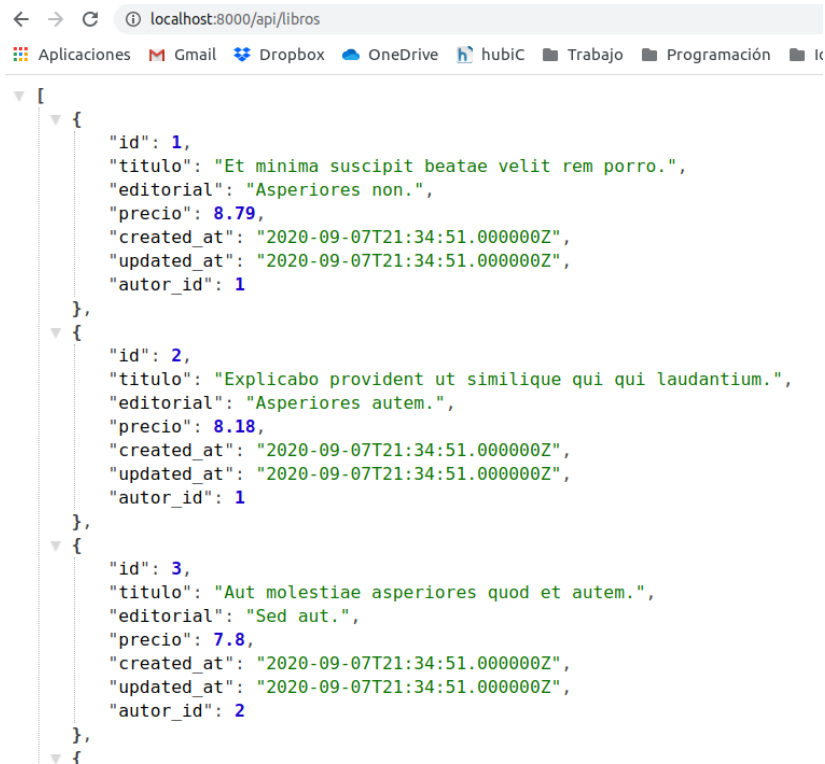
```
+-----+
| Method      | URI                      | Name              |
+-----+
| GET|HEAD    | api/libros               | libros.index      |
| POST        | api/libros               | libros.store       |
| GET|HEAD    | api/libros/{libro}       | libros.show       |
| PUT|PATCH  | api/libros/{libro}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

2.3. 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.



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 sesión 4, o si usamos el parámetro `--model` para asociarlo, como hemos hecho aquí.

2.3.1. 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:

1. 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'];
```

2. 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
    ];
}
```

3. 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);
}
```

2.4. 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.

2.3.1. 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)
{
    ...
}
```

2.3.2. Respuestas de error

Por otra parte, debemos asegurarnos de que cualquier error que se produzca en la parte del 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**, 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.

3. Probando los servicios con Postman

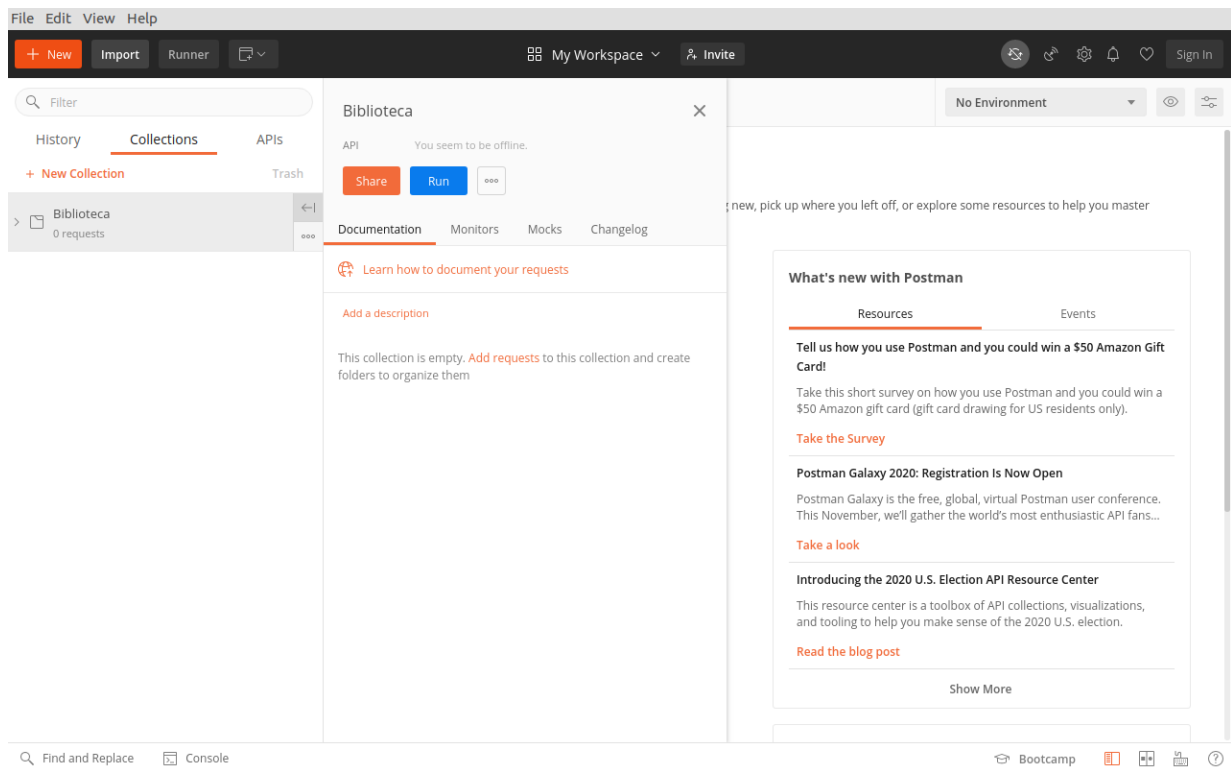
Ya hemos visto que probar unos servicios de listado (GET) es sencillo a través de un navegador. Pero los servicios de inserción (POST), modificación (PUT) o borrado (DELETE) exigen de otras herramientas para poder ser probados. Podríamos definir formularios con estos métodos encapsulados, pero el esfuerzo de definir esos formularios para luego no utilizarlos más no merece mucho la pena. Veremos a continuación una herramienta muy útil para probar todo tipo de servicios sin necesidad de implementar nada adicional.

Postman es una aplicación gratuita y multiplataforma que permite enviar todo tipo de peticiones a un servidor determinado, y examinar la respuesta que éste produce. De esta forma, podemos comprobar que los servicios ofrecen la información adecuada antes de ser usados por una aplicación cliente real.

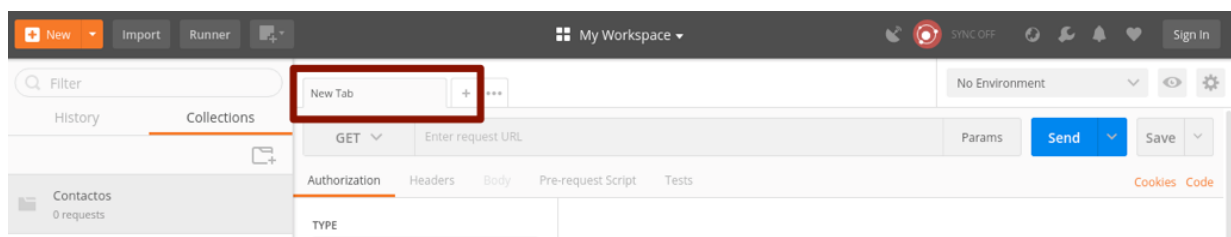
Para descargar e instalar Postman, debemos ir a su web oficial, a la [sección de descargas](#), y descargar la aplicación (en el caso de nuestra máquina virtual, descargaremos la versión para Linux de 64 bits). Es un archivo portable, que se descomprime y dentro está la aplicación lista para ejecutarse.

La primera vez que la ejecutemos nos preguntará si queremos registrarnos, de forma que podamos compartir los proyectos que hagamos entre los distintos equipos en que estemos registrados, pero podemos saltar este paso haciendo clic en el enlace inferior.

Tras iniciar la aplicación, veremos la pantalla de inicio de Postman. En un principio aparecerán varias opciones en la zona central, para crear colecciones o peticiones, aunque también las podemos crear desde el botón *New* en la esquina superior izquierda. Por ejemplo, podemos crear una colección "Biblioteca", y aparecerá en el panel izquierdo:



Desde el mismo botón *New* en la esquina superior izquierda podemos crear nuevas peticiones y asociarlas a una colección. Existe una forma alternativa (quizá más cómoda) de crear esas peticiones, a través del panel de pestañas, añadiendo nuevas:



3.1. Añadir peticiones GET

Para añadir una petición, habitualmente elegiremos el tipo de comando bajo las pestañas (GET, POST, PUT, DELETE) y la URL asociada a dicho comando. Por ejemplo:

The screenshot shows the Postman interface. At the top, there's a status bar with 'GET localhost:8000/api/libros' and a 'No Environment' dropdown. Below this, the 'Untitled Request' section shows the method 'GET' and the URL 'localhost:8000/api/libros'. To the right are 'Send', 'Save', and 'Build' buttons. Below the URL bar are tabs for 'Params', 'Authorization', 'Headers (6)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. The 'Params' tab is active, showing a table for 'Query Params' with columns 'KEY', 'VALUE', and 'DESCRIPTION'. There are also links for 'Cookies' and 'Code'.

Entonces, podemos hacer clic en el botón "Save" en la parte derecha, y guardar la petición para poderla reutilizar. Al guardarla, nos pedirá que le asignemos un nombre (por ejemplo, "GET libros" en este caso), y la colección en la que se almacenará (nuestra colección de "Biblioteca").

The 'SAVE REQUEST' dialog is shown. It explains that requests are saved in collections. The 'Request name' field contains 'GET libros'. The 'Request description (Optional)' field has a placeholder text. Below this, it says 'Descriptions support Markdown'. A section 'Select a collection or folder to save to:' shows a search bar and a list with 'Biblioteca' selected. At the bottom are 'Cancel' and 'Save to Biblioteca' buttons.

Después, podremos ver la prueba asociada a la colección, en el panel izquierdo:

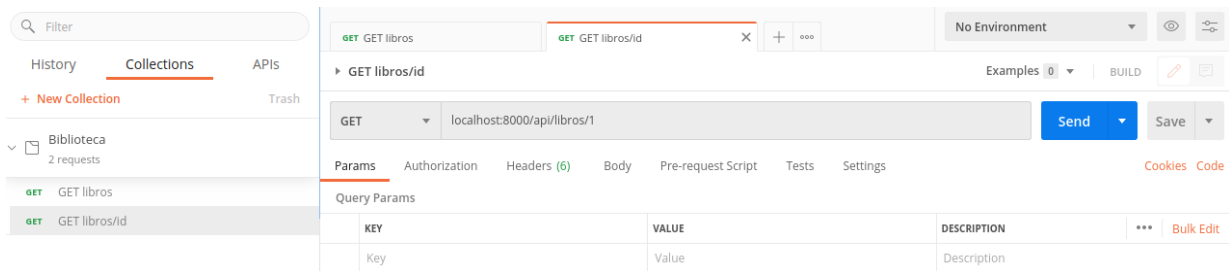
The screenshot shows the Postman interface with the 'Biblioteca' collection selected in the left sidebar. The collection contains one request, 'GET GET libros'. The right panel shows the details of this request, including the method 'GET' and the URL 'localhost:8000/api/libros'. The 'Params' tab is active, showing a table for 'Query Params' with columns 'KEY' and 'VALUE'.

Si seleccionamos esta prueba y pulsamos en el botón azul de "Send" (parte derecha), podemos ver la respuesta emitida por el servidor en el panel inferior de respuesta (si tenemos, claro está, el servidor en marcha):

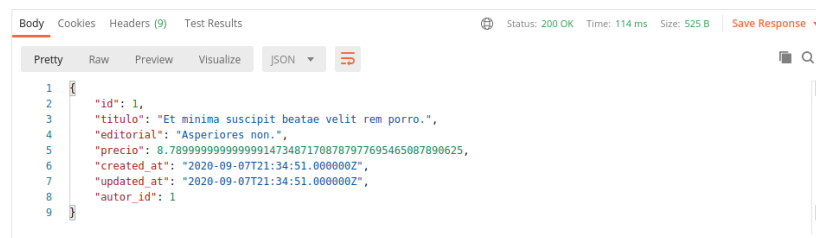


```
1 {
2   "id": 1,
3   "titulo": "Et minima suscipit beatae velit rem porro.",
4   "editorial": "Asperiores non.",
5   "precio": 8.78999999999999914734871708787977695465087890625,
6   "created_at": "2020-09-07T21:34:51.000000Z",
7   "updated_at": "2020-09-07T21:34:51.000000Z",
8   "autor_id": 1
9 },
```

Siguiendo estos mismos pasos, podemos también crear una nueva petición para obtener un libro a partir de su *id*, por GET:



Bastaría con reemplazar el *id* de la URL por el que queramos consultar realmente. Si probamos esta petición, obtendremos la respuesta correspondiente:

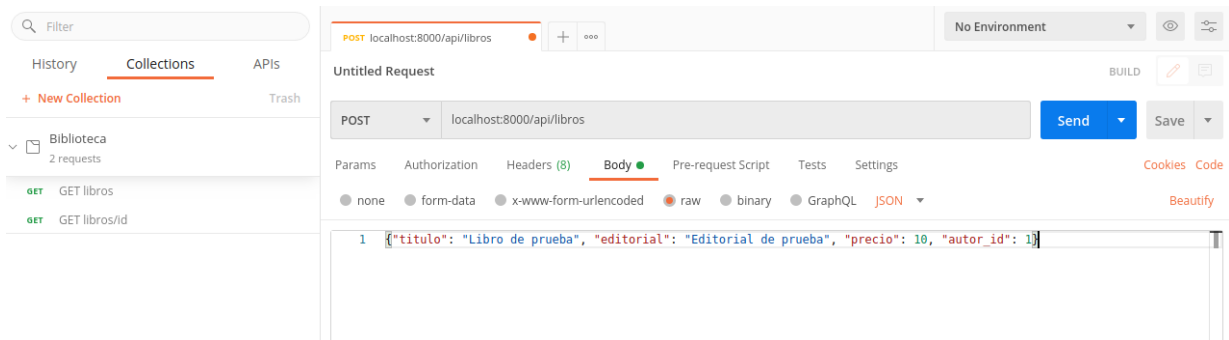


```
1 {
2   "id": 1,
3   "titulo": "Et minima suscipit beatae velit rem porro.",
4   "editorial": "Asperiores non.",
5   "precio": 8.78999999999999914734871708787977695465087890625,
6   "created_at": "2020-09-07T21:34:51.000000Z",
7   "updated_at": "2020-09-07T21:34:51.000000Z",
8   "autor_id": 1
9 }
```

3.2. Añadir otros tipos de peticiones (POST, PUT, DELETE)

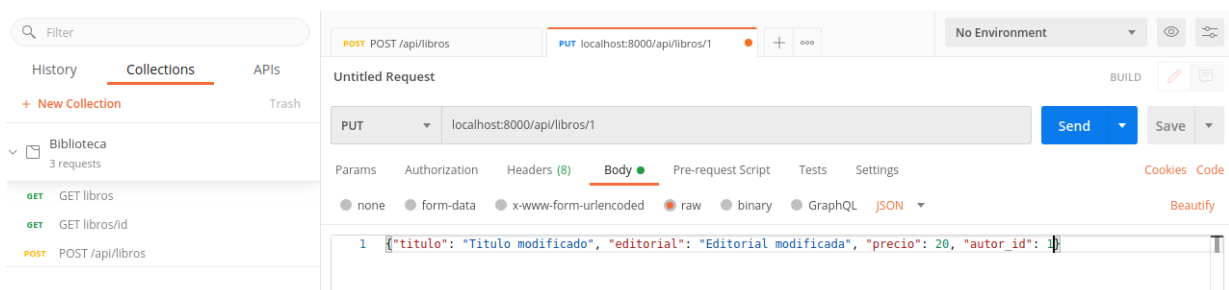
Las peticiones **POST** difieren de las peticiones GET en que se envía cierta información en el cuerpo de la petición. Esta información normalmente son los datos que se quieren añadir en el servidor. ¿Cómo podemos hacer esto con Postman?

En primer lugar, creamos una nueva petición, elegimos el comando POST y definimos la URL (en nuestro caso, *localhost:8000/api/libros* o algo similar, dependiendo de cómo tengamos en marcha el servidor). Entonces, hacemos clic en la pestaña *Body*, bajo la URL, y establecemos el tipo como *raw* para que nos deje escribirlo sin restricciones. También conviene cambiar la propiedad *Text* para que sea *JSON*, y que así el servidor recoja el tipo de dato adecuado. Se añadirá automáticamente una cabecera de petición (Header) que especificará que el tipo de contenido que se va a enviar son datos JSON. Después, en el cuadro de texto bajo estas opciones, especificamos el objeto JSON que queremos enviar para insertar:

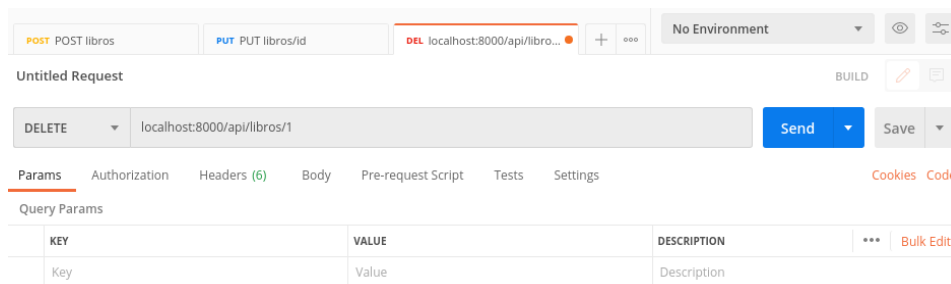


Tras esto, basta con guardar la petición como hemos hecho con las anteriores, y lanzarla para ver el resultado.

En cuanto a las peticiones **PUT**, procederemos de forma similar a las peticiones POST: debemos elegir el comando (PUT en este caso), la URL, y completar el cuerpo de la petición con los datos que queramos modificar del contacto. En este caso, además, el *id* del libro lo enviaremos también en la propia URL:

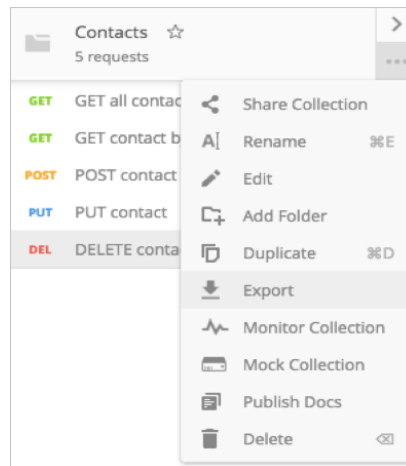


Para peticiones **DELETE**, la mecánica es similar a la ficha del elemento (operación GET), cambiando el comando GET por DELETE, y sin necesidad de establecer nada en el cuerpo de la petición:



3.3. Exportar/Importar colecciones

Podemos exportar e importar nuestras colecciones en Postman, de forma que podemos llevarlas de un equipo a otro. Para **exportar** una colección, hacemos clic en el botón de puntos suspensivos (...) que hay junto a ella en el panel izquierdo, y elegimos *Export*.



Nos preguntará para qué versión de Postman queremos exportar (normalmente la recomendada es la mejor opción). Se creará un nuevo archivo Postman en la ubicación que elijamos.

Si queremos **importar** una colección previamente exportada, podemos hacer clic en el botón *Import* de la esquina superior izquierda en la ventana principal y elegir después el archivo a importar.

En este punto, puedes realizar el [Ejercicio 1](#) de los propuestos al final de la sesión.

4. Autenticación en servicios REST

En una API REST también puede ser necesario proteger ciertos servicios, de forma que sólo puedan acceder a ellos los usuarios autenticados. Sin embargo, en este caso no tenemos disponible el mecanismo de autenticación basado en sesiones que vimos en temas anteriores, ya que la parte cliente que consula la API REST no tiene por qué estar basada en un navegador. Podríamos acceder desde una aplicación de escritorio hecha en Java, por ejemplo, o desde una aplicación móvil, y en estos casos no podríamos disponer de las sesiones, propias de clientes web o navegadores. En su lugar, emplearemos un mecanismo de autenticación basado en **tokens**.

4.1. Fundamentos de la autenticación basada en tokens

La autenticación basada en tokens es un mecanismo de validación de usuarios en aplicaciones cliente-servidor que podríamos decir que es más universal que la autenticación basada en sesiones, ya que permite autenticar usuarios provenientes de distintos tipos de clientes. Lo que se hace es lo siguiente:

- El usuario necesita enviar sus credenciales (*login* y *password*), de forma similar a como se hace en una aplicación web normal, aunque esta vez los datos se envían normalmente en formato JSON.
- El servidor valida esas credenciales y, si son correctas, genera una cadena de texto llamada *token*, de una cierta longitud, y que servirá para identificar unívocamente al usuario a partir de ese momento. Dicho *token* debe ser enviado de vuelta (también en formato JSON) al cliente que se validó
- A partir de este punto, el cliente debe adjuntar el *token* como parte de la información en cada petición que realiza a una zona de acceso restringido, de forma que el servidor pueda consultar el token y comprobar si corresponde con el de algún usuario autorizado. Este token normalmente se envía en una

cabecera de la petición llamada *Authorization*, como veremos después, y el servidor puede consultar el valor de dicha cabecera para verificar el acceso del cliente.

4.2. Alternativas para la implementación de la autenticación basada en tokens

Podemos emplear distintas alternativas para la autenticación basada en tokens bajo Laravel. Comentaremos en esta sesión dos de ellas.

- Por un lado, podemos emplear el mecanismo nativo de Laravel para autenticación basada en tokens. Como ventajas principales, no se necesita instalar ninguna dependencia adicional, y es relativamente sencillo de utilizar. Como inconvenientes, requiere añadir un campo más a la tabla de usuarios, para almacenar el token generado para cada usuario, y requiere también de una gestión manual del token, aunque es sencilla.
- Por otro lado podemos valernos de la librería [Laravel Sanctum](#), que proporciona mecanismos de autenticación para APIs y para SPAs (*Single Page Applications*, aplicaciones de página única). Entre sus ventajas podemos destacar que es sencilla de integrar en la aplicación y automatiza algunos aspectos de la gestión de tokens, además de contar con el soporte oficial de Laravel. Como inconvenientes, es una librería más intrusiva que la anterior, ya que requiere crear una tabla adicional donde almacenar los tokens.

En los siguientes apartados veremos cómo proteger mediante tokens un proyecto sencillo en Laravel empleando cada uno de estos mecanismos. Como ejercicio de este apartado se pide que elijáis cualquiera de ellos y sigáis paso a paso el ejemplo para configurar la protección mediante tokens en él.

4.2.1. Preparando el ejemplo base

Partiremos de un mismo proyecto base, que luego adaptaremos en función del mecanismo de autenticación elegido. Comenzaremos creando un proyecto llamado `pruebaToken`, en nuestra carpeta de proyectos:

```
laravel new pruebaToken
```

Después, eliminaremos las migraciones que no vamos a utilizar de la carpeta `database/migrations`: en concreto, eliminaremos los archivos sobre `create_password_resets_table` y `create_failed_jobs_table`, y dejaremos sólo la migración de la tabla de usuarios. Sobre ella, editaremos los métodos `up` y `down` para dejar sólo los campos que nos interesen, y renombrar la tabla a *usuarios*, de este modo:

```
public function up()
{
    Schema::create('usuarios', function (Blueprint $table) {
        $table->id();
        $table->string('login')->unique;
        $table->string('password');
        $table->timestamps();
    });
}
...
public function down()
{
    Schema::dropIfExists('usuarios');
}
```

A continuación, renombramos el modelo `App\Models\User.php` a `App\Models\Usuario.php`, cambiando también el nombre de la clase interior:

```
class Usuario extends Authenticatable
{
    use HasFactory, Notifiable;

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name',
        'password',
    ];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = [
        'password',
    ];
}
```

Y hacemos lo mismo con el *factory* y el *seeder* correspondiente:

```
namespace Database\Factories;

use App\Models\Usuario;
use Illuminate\Database\Eloquent\Factories\Factory;

class UsuarioFactory extends Factory
{
    /**
     * The name of the factory's corresponding model.
     *
     * @var string
     */
    protected $model = Usuario::class;

    /**
     * Define the model's default state.
     *
     * @return array
     */
    public function definition()
    {
        return [
            'login' => $this->faker->word,
            'password' => bcrypt('1234')
        ];
    }
}
```

```
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        \App\Models\Usuario::factory(2)->create();
    }
}
```

Vamos a modificar también el archivo `.env` del proyecto para acceder a una base de datos llamada `pruebaToken`, que deberemos crear a través de *phpMyAdmin*:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=pruebaToken
DB_USERNAME=root
DB_PASSWORD=
```

Necesitamos también editar el archivo `App\Http\Middleware\Authenticate.php` para indicar en el método `redirectTo` que sólo queremos redirigir al formulario de login cuando la petición no espere una respuesta en formato JSON. En caso contrario, no hay que mostrar dicho formulario, sino enviar una respuesta JSON adecuada. De hecho, si la aplicación sólo va a tener servicios REST podríamos eliminar o dejar comentado el código de este método para que no trate de redirigir a ningún formulario.

```
class Authenticate extends Middleware
{
    protected function redirectTo($request)
    {
        /*
        if (! $request->expectsJson()) {
            return route('login');
        }
        */
    }
}
```

Por otra parte, debemos editar el archivo `App\Exceptions\Handler.php`, en concreto su método `register` para definir los diferentes errores que pueden producirse y los mensajes que hay que devolver en cada caso:

```

use Illuminate\Auth\AuthenticationException;
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;
use Illuminate\Database\Eloquent\ModelNotFoundException;
use Illuminate\Validation\ValidationException;
use Throwable;

class Handler extends ExceptionHandler
{
    ...
    public function register()
    {
        $this->renderable(function (Throwable $exception) {
            if (request()->is('api*'))
            {
                if ($exception instanceof ModelNotFoundException)
                    return response()->json(
                        ['error' => 'Elemento no encontrado'], 404);
                else if ($exception instanceof AuthenticationException)
                    return response()->json(
                        ['error' => 'Usuario no autenticado'], 401);
                else if ($exception instanceof ValidationException)
                    return response()->json(
                        ['error' => 'Datos no válidos'], 400);
                else if (isset($exception))
                    return response()->json(
                        ['error' => 'Error en la aplicación (' .
                            get_class($exception) . '):' .
                            $exception->getMessage()], 500);
            }
        });
    }
}

```

Finalmente, vamos a definir un controlador de API con una serie de métodos de prueba. No lo vamos a vincular a ningún modelo, porque generaremos unos datos a mano en cada método para simplificar el código. Escribimos este comando:

```
php artisan make:controller Api/PruebaController --api
```

Rellenamos el código de los métodos del controlador con alguna respuesta sencilla para cada caso:

```
class PruebaController extends Controller
{
    public function index()
    {
        return response()->json(['mensaje' => 'Accediendo a index']);
    }

    public function store(Request $request)
    {
        return response()->json(['mensaje' => 'Insertando'], 201);
    }

    public function show($id)
    {
        return response()->json(['mensaje' => 'Ficha de ' . $id]);
    }

    public function update(Request $request, $id)
    {
        return response()->json(['mensaje' => 'Actualizando elemento']);
    }

    public function destroy($id)
    {
        return response()->json(['mensaje' => 'Borrando elemento']);
    }
}
```

Y añadimos las rutas correspondientes en el archivo `routes/api.php`:

```
Route::apiResource('prueba', PruebaController::class);
```

A partir de este punto, vamos a proteger el acceso a alguno de estos métodos. Escoge uno de los siguientes apartados (4.3 o 4.4) para definir el mecanismo de autenticación basado en tokens correspondiente. También puedes intentar hacerlos todos; en este caso, copia y pega otra vez el proyecto Laravel, para trabajar por separado en cada carpeta con un mecanismo diferente.

4.3. Autenticación basada en tokens nativa

Vamos a emplear en esta sección la autenticación nativa por tokens que ofrece Laravel. Los pasos a seguir los indicamos a continuación.

4.3.1. Configuración básica

En primer lugar, modificamos la migración de la tabla de usuarios para añadir un nuevo campo donde almacenar el token. Dicho campo basta con que tenga 60 caracteres de longitud, y será necesario también que sea único para cada usuario:

```
public function up()
{
    Schema::create('usuarios', function (Blueprint $table) {
        $table->id();
        $table->string('login')->unique;
        $table->string('password');
        $table->string('api_token', 60)->unique()->nullable();
        $table->timestamps();
    });
}
```

Podemos lanzar ya la migración para que se cree la tabla y se rellene con los usuarios que hayamos indicado en el *seeder*.

```
php artisan migrate:fresh --seed
```

También debemos modificar el archivo `config/auth.php` para indicar cuál es el modelo de usuarios que vamos a utilizar:

```
'providers' => [
    'users' => [
        'driver' => 'eloquent',
        'model' => App\Models\Usuario::class,
    ],
],
```

En este mismo fichero, también podemos modificar el *guard* por defecto, que es *web*, para que sea *api*, si nuestra aplicación no va a tener autenticación web:

```
'defaults' => [
    'guard' => 'api',
    ...
],
```

4.3.2. Protección de rutas

Para proteger las rutas de acceso restringido, primero crearemos un controlador que se encargue de validar las credenciales del usuario:

```
php artisan make:controller Api/LoginController
```

Definimos un método `login`, por ejemplo, que validará las credenciales que le lleguen (login y password). Si son correctas, generará una cadena de texto aleatoria de 60 caracteres y la almacenará en el campo `api_token` del usuario validado. También devolverá dicho token como respuesta en formato JSON. En caso de que haya un error en la autenticación, enviará de vuelta un mensaje de error, con el código 401 de acceso no autorizado.

```
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Str;
use Illuminate\Support\Facades\Hash;
use App\Models\Usuario;

class LoginController extends Controller
{
    public function login(Request $request)
    {
        $usuario = Usuario::where('login', $request->login)->first();

        if (!$usuario ||
            !Hash::check($request->password, $usuario->password))
        {
            return response()->json(
                ['error' => 'Credenciales no válidas'], 401);
        }
        else
        {
            $usuario->api_token = Str::random(60);
            $usuario->save();
            return response()->json(['token' => $usuario->api_token]);
        }
    }
}
```

Definimos en el archivo `routes/api.php` una ruta que redirija a este método, para cuando el usuario quiera autenticarse (recuerda añadir con `use` la correspondiente clase):

```
Route::post('login', [LoginController::class, 'login']);
```

También podemos eliminar en este caso la ruta predefinida de este archivo, que emplea la autenticación nativa de Laravel:

```
// Eliminar esta ruta:  
Route::middleware('auth:api')->get('/user', function (Request $request) {  
    return $request->user();  
});
```

Para proteger las rutas que necesitemos en los controladores API, las especificamos en el constructor del controlador. Por ejemplo, así protegeríamos todas las rutas de nuestro controlador `PruebaController`, salvo `index` y `show`:

```
class PruebaController extends Controller  
{  
    public function __construct()  
    {  
        $this->middleware('auth:api',  
            ['except' => ['index', 'show']]);  
    }  
    ...  
}
```

Alternativamente, también podemos emplear el modificador `only` en lugar de `except` para indicar las rutas concretas que queremos proteger.

Con esto ya tenemos el mecanismo de autenticación por token establecido, y las rutas protegidas. Echa un vistazo al apartado 4.5 para ver cómo probarlo todo desde Postman.

4.4. Autenticación basada en tokens usando Laravel Sanctum

Como hemos comentado anteriormente, Laravel Sanctum es una librería que proporciona mecanismos de autenticación para SPAs (*Single Page Applications*, aplicaciones de página única), y APIs. En nuestro caso, la emplearemos para autenticarnos mediante tokens en nuestras APIs. Los pasos a seguir para la configuración son los siguientes...

4.4.1. Configuración de Sanctum

En primer lugar, debemos incorporar Laravel Sanctum a nuestro proyecto, escribiendo este comando desde la raíz del mismo:

```
composer require laravel/sanctum
```

Después, debemos publicar la configuración de Sanctum y su fichero de migración, que generará una tabla adicional donde almacenar los tokens. Escribimos el siguiente comando (todo en una línea, aunque aquí se divide en dos para poderlo ver completo):

```
php artisan vendor:publish  
--provider="Laravel\Sanctum\SanctumServiceProvider"
```

Al finalizar este paso, tendremos la migración creada y un archivo de configuración `config/sanctum.php` disponible, para editar la configuración por defecto de la librería. Por ejemplo, podemos editarlo para especificar el tiempo de vida (TTL) de los tokens. El siguiente ejemplo establece un tiempo de vida de 5 minutos, por ejemplo, aunque en el caso de aplicaciones basadas en tokens es habitual dejar tiempos mucho mayores (o indefinidos, según el caso, dejando esta propiedad a `null`):

```
'expiration' => 5,
```

Después, debemos lanzar la migración que se ha creado, junto con las que tengamos pendientes (la de la tabla de usuarios, por ejemplo). Se añadirá una tabla llamada *personal_access_tokens* a nuestra base de datos.

```
php artisan migrate:fresh --seed
```

Finalmente, debemos modificar el modelo de usuarios (`App\Models\Usuario`) para añadir el *trait* `HasApiTokens` . De este modo se vincula el modelo de usuario con los tokens que se vayan a generar para los mismos.

```
...  
use Laravel\Sanctum\HasApiTokens;  
  
class Usuario extends Authenticatable  
{  
    use HasApiTokens, HasFactory, Notifiable;  
    ...  
}
```

4.4.2. Protección de rutas

Para proteger las rutas de acceso restringido, primero crearemos un controlador que se encargue de validar las credenciales del usuario:

```
php artisan make:controller Api/LoginController
```

Definimos un método `login`, por ejemplo, que validará las credenciales que le lleguen (login y password). Si son correctas, llamará al método `createToken` de Sanctum (incorporado al usuario a través del *trait* `HasApiTokens`), asociándolo al login del usuario entrante, y le devolverá el token en formato texto plano, como un objeto JSON. En caso de que haya un error en la autenticación, enviará de vuelta un mensaje de error, con el código 401 de acceso no autorizado.

```
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use App\Models\Usuario;

class LoginController extends Controller
{
    public function login(Request $request)
    {
        $usuario = Usuario::where('login', $request->login)->first();

        if (!$usuario ||
            !Hash::check($request->password, $usuario->password))
        {
            return response()->json(
                ['error' => 'Credenciales no válidas'], 401);
        }
        else
        {
            return response()->json(['token' =>
                $usuario->createToken($usuario->login)->plainTextToken]);
        }
    }
}
```

Definimos en el archivo `routes/api.php` una ruta que redirija a este método, para cuando el usuario quiera autenticarse (recuerda añadir con `use` la correspondiente clase):

```
Route::post('login', [LoginController::class, 'login']);
```

También podemos eliminar en este caso la ruta predefinida de este archivo, que emplea la autenticación nativa de Laravel:

```
// Eliminar esta ruta:  
Route::middleware('auth:api')->get('/user', function (Request $request) {  
    return $request->user();  
});
```

Para proteger las rutas que necesitemos en los controladores API, las especificamos en el constructor del controlador. Por ejemplo, así protegeríamos todas las rutas de nuestro controlador `PruebaController`, salvo `index` y `show`:

```
class PruebaController extends Controller  
{  
    public function __construct()  
    {  
        $this->middleware('auth:sanctum',  
            ['except' => ['index', 'show']]);  
    }  
    ...  
}
```

Alternativamente, también podemos emplear el modificador `only` en lugar de `except` para indicar las rutas concretas que queremos proteger.

Con esto ya tenemos el mecanismo de autenticación por token establecido, y las rutas protegidas. Echa un vistazo al apartado 4.5 para ver cómo probarlo todo desde Postman.

4.5. Prueba de autenticación en Postman

Veamos cómo probar la autenticación por token en el proyecto que acabamos de realizar, por cualquiera de los métodos vistos antes.

En primer lugar, y tras poner en marcha el proyecto, vamos a asegurarnos de que podemos acceder sin restricciones a los dos servicios que no requieren autorización (`index` o `show`). Por ejemplo podemos hacer una petición como esta, y ver que obtenemos el resultado devuelto por el método `index`:

GET localhost:8000/api/prueba

Untitled Request

GET localhost:8000/api/prueba

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE
Key	Value

Body Cookies Headers (10) Test Results

Pretty Raw Preview Visualize JSON

```
1 {  
2   "mensaje": "Accediendo a index"  
3 }
```

Ahora vamos a intentar acceder a un recurso protegido, como por ejemplo el borrado, y veremos que obtenemos un código 401 indicando que no nos hemos autenticado:

DELETE localhost:8000/api/prueba/1

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize

Status: 401 Unauthorized

```
{ "error": "Usuario no autenticado" }
```

Para autenticarnos con nuestras credenciales y obtener el token, enviamos una petición POST al servicio `api/login`, enviando por JSON nuestro login y password. Obtendremos de vuelta el token en formato JSON:

POST localhost:8000/api/login

Params Authorization Headers (8) Body Pre-request Script Tests Settings

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

```
1 { "login": "sint", "password": "1234" }
```

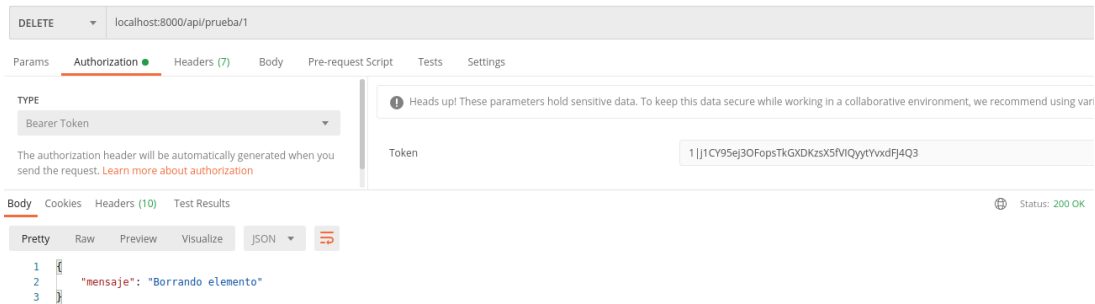
Body Cookies Headers (10) Test Results

Pretty Raw Preview Visualize JSON

```
1 {  
2   "token": "1jj1CY9Sej30FopsTk6XDKzsX5fVIQyytYvxdFJ4Q3"  
3 }
```

Ahora debemos copiar este token, y pegarlo en la petición de acceso restringido. Deberemos pegarlo en la cabecera *Authorization* (abrir esa pestaña bajo la URL de la petición en Postman), y lo normal es enviarlo

como un *Bearer token*, según los estándares. Entonces sí tendremos la respuesta correcta de la operación solicitada.



A la hora de trasladar estas pruebas a una aplicación "real", enviaríamos las credenciales por JSON al servidor, obtendríamos el token de vuelta y lo almacenaríamos localmente en alguna variable o soporte (por ejemplo, en el elemento `localStorage`, si trabajamos con algún framework JavaScript). Después, ante cada petición JSON que hiciéramos al servidor, adjuntaríamos este token en la cabecera *Authorization* para que fuese validado por el servidor.

5. Ejercicios propuestos

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.
 - `put` 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 Postman 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.

Ejercicio 2

Como último ejercicio de esta sesión, se pide que sigáis los pasos indicados en el punto 4, donde se explica cómo proteger aplicaciones basadas en servicios REST mediante tokens. En el punto 4.2 se indica que existen distintas formas de hacerlo, y en los puntos 4.3 y 4.4 se detallan dos de ellas. Lo que se pide como ejercicio es que sigáis los pasos detallados en cualquiera de las dos opciones (4.3 o 4.4) para proteger el proyecto de ejemplo que se crea en ese apartado.

Deberéis adjuntar como resultado de este ejercicio el proyecto **pruebaToken** con el código completo para proteger mediante tokens las rutas restringidas del controlador `PruebaController` que habremos definido. También debéis adjuntar la colección de pruebas de Postman que habréis empleado para probar la seguridad de la aplicación.

¿Qué entregar?

Como entrega de esta sesión deberás adjuntar lo siguiente:

- Comprimir el proyecto **blog** con todos los cambios incorporados del Ejercicio 1, y eliminando las carpetas `vendor` y `node_modules` como se explicó en las sesiones anteriores. Añade dentro también la colección Postman para probar los servicios. Renombra el archivo comprimido a `blog_08.zip`.
- Comprimir el proyecto `pruebaToken` con el proyecto del Ejercicio 2 siguiendo uno de los dos mecanismos de autenticación propuestos. Elimina también la carpeta `vendor`, y añade dentro del ZIP la colección Postman para probar los servicios.

Entrega ambos archivos por separado en la tarea de la sesión, o bien comprimidos los dos en otro archivo ZIP llamado `session8.zip`.