Introducción a los 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. 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.

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.

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}]
```