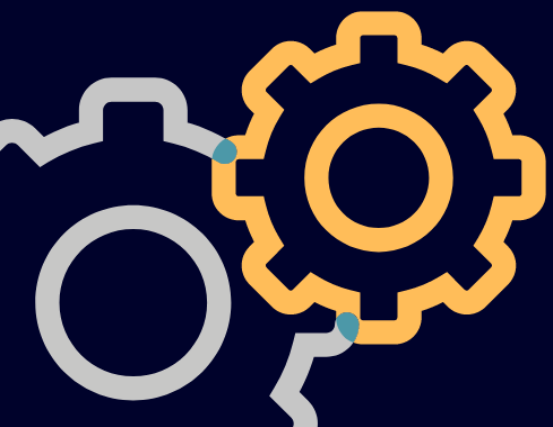
APRENDE A TRABAJAR CON WEB APIS



Alberto Ramírez Caballero
Versión 1.2

Email de contacto: alberto.r.caballero.87@gmail.com
Twitter: https://twitter.com/backalber
Mi sitio web: https://cosasdedevs.com

Índice

¿Qué voy a aprender aquí?

En esta guía **aprenderás a entender el funcionamiento de una Web API**, te facilitará el trabajo cuando la documentación de una API sea escasa o inexistente y tengas problemas para comunicarte con ella.

También **obtendrás los conocimientos para crear una Web API siguiendo la arquitectura REST** y vas a ver todo el ciclo desde que realizas una petición hasta la respuesta.

No importa si eres Frontend, Backend o Fullstack, ya que entender todo el ciclo y su construcción te ayudará sea cual sea tu posición.

Mi idea es ir bastante al grano y no enredarme con explicaciones confusas e innecesarias que añadan una complejidad extra.

Conforme vayas leyendo esta guía, podrás observar que en paréntesis escribo algunos nombres en inglés. Eso es porque muchas veces vas a encontrar los nombres en este idioma, incluso leyendo textos en español, así que los voy a añadir también para que más adelante los puedas identificar sin problema.

En esta guía no solo vamos a hablar de Web APIs, sino que vamos a ver todas las partes implicadas como la **URL**, el **protocolo HTTP**, el funcionamiento del ciclo desde que realizamos **una petición hasta la respuesta**, **arquitectura REST** y al final aprenderemos como construir nuestra propia Web API.

Y dicho esto, ¡Al lío!

¿Qué es una API?

Es la abreviatura de Application Programming Interfaces, que en español sería interfaz de programación de aplicaciones. Te has quedado igual, ¿No? A mí también me pasó cuando lo leí por primera vez, así que te voy a dar una definición más “amigable”.

Una API es **una aplicación con la cual nos podemos comunicar para poder obtener información o realizar ciertas acciones**.

Esta **API puede ser un web service o no**. Un ejemplo de API que **NO es un web service**, es la API de Android. Cuando queramos construir una aplicación para este SO, podremos valernos de su API para acceder a funcionalidades del sistema como puede ser la cámara de un Smartphone.

En esta guía vamos a enfocarnos en las Web APIs así que a partir de ahora, cuando utilice el término API, me estaré refiriendo a este tipo de servicios.

¿Qué información podemos obtener de una API?

La que queramos mostrar en la API, como por ejemplo, obtener información de una base de datos, hacer de puente entre otra API y retornar su información, etc.

¿Qué acciones podemos efectuar?

Como en el caso anterior, el límite lo ponemos al crear la API. Por ejemplo, insertar, actualizar y borrar información de una base de datos.

Ejemplo con el que trabajaremos en la guía

Para explicar varios puntos de esta guía, vamos a trabajar sobre un ejemplo de una API con la que basarnos.

La API en cuestión será una herramienta en la que podremos gestionar un listado de tareas propio. Para administrar dicha lista, primero deberemos registrarnos como usuario y una vez hecho podremos **hacer login** en la API. Al realizar esta acción recibiremos un **token** de autenticación que servirá para poder identificarnos como nuestro usuario en la API y este token nos permitirá crear y gestionar **nuestras tareas**.

Cada tarea estará relacionada con el usuario que la ha generado y cada usuario solo podrá ver y gestionar sus tareas.

También tenemos un usuario administrador que podrá ver el listado de usuarios y además visualizar las tareas de los demás usuarios.

Por último, todas las respuestas de nuestra API se retornarán en formato **JSON**.

¿Qué es una URL?

Una URL es una dirección única que se utiliza para acceder a una parte de una web. Si accedes a la siguiente URL:

```
https://cosasdedevs.com/posts/retornar-html-fastapi/
```

Vas a visualizar uno de mis posts. Las tripas de esta página (el HTML) además **contiene accesos a más URLs** de mi web como la imagen de cabecera, archivos CSS y JS que hacen que la página web se visualice de la manera en la que la ves.

Partes de una URL

Una URL está compuesta de varias partes, unas son obligatorias y otras pueden ser opcionales. En esta guía vamos a explicar las más importantes.

Protocolo

Indica que protocolo debe usar el navegador para procesar la información que nos devolverá la URL. **Para web, los protocolos son HTTP y HTTPS**. De ellos hablaremos más adelante:

```
https://cosasdedevs.com/posts/retornar-html-fastapi/
```

Dominio

Es la dirección de un sitio web. **Esta dirección identifica un sitio web en concreto y es único**, por lo que no puede ser replicado. En este caso el dominio cosasdedevs.com me pertenece y haciendo que

apunte a un servidor donde tengo alojada mi web, podéis ver todo su contenido si accedes a él desde un navegador.

```
https://www.cosasdedevs.com/posts/retornar-html-fastapi/
```

O

```
https://cosasdedevs.com/posts/retornar-html-fastapi/
```

Ruta (path)

Es la ruta a un recurso en concreto de una web. En el siguiente caso es la ruta a uno de mis posts, pero también puede ser a una imagen, CSS, JS, etc.

```
https://cosasdedevs.com/posts/retornar-html-fastapi/
```

Las rutas de una API también son conocidas como endpoints.

API Endpoint

Un endpoint es la ruta final hacia un recurso. En nuestra API de ejemplo, un endpoint podría ser la ruta que obtiene todas las tareas. Para ello podríamos definir el endpoint “tareas”:

```
https://ejemplo.com/tareas
```

Tipos de parámetros que puede recibir una URL

En una web o API podemos permitir enviar parámetros vía URL. Estos nos pueden servir para modificar su comportamiento, guardar cierta

información que nos pueda ser relevante o cualquier uso que le quieras dar. Eso si, **tu web o API deberá estar preparada para recibir esos parámetros**, ya que si no es así no servirán absolutamente de nada.

Parámetros de consulta (query params)

Estos parámetros no pertenecen a la URL y se asignan en formato **clave=valor**. Para añadir estos parámetros, debemos escribir el signo "?" al final de la URL y después añadir el nombre del parámetro, el símbolo de = y su valor. Si queremos añadir más parámetros, estos deberán ir separados entre sí por el signo "&" en cada par clave=valor:

```
https://cosasdedevs.com/posts/enviar-parametros-url-path-fastapi/?param1=hola&param2=que_tal&param3=adios
```

Para ver un caso de uso vamos a utilizar nuestro ejemplo de API. En él, podemos obtener un listado con las tareas, pero **imagina que empezamos a manejar una cantidad desproporcionada** y cada vez es más difícil trabajar con toda esa información.

Para solucionar esto, puedes añadir un límite de tareas que quieres que devuelva por petición y además, si quieres tener la opción de modificarlo, podemos definir un parámetro de consulta al que llamaremos "**limit**" (límite) que al enviarlo nos permita cambiar el número de tareas que recibimos en una petición.

Al añadir un límite nuestra respuesta pasa a ser paginada. Esto quiere decir que si tenemos 100 tareas y un límite de 10 tareas por petición vamos a necesitar un nuevo parámetro al que llamaremos

"**page**" (página). Este parámetro será numérico. Si no se envía podemos darle el valor de 1 e indicará que rango que tareas queremos obtener. Ejemplo:

Página 1: De la tarea 1 a la 10.
Página 2: De la tarea 11 a la 20.
Página 3: De la tarea 21 a la 30.
Y así sucesivamente hasta obtener las 100.

Otro ejemplo de parámetro que podríamos crear en nuestra API sería un filtro para obtener las tareas realizadas o las que están sin hacer, al que llamaremos "**is_done**" (si está hecha). Si lo enviamos e indicamos "**true**" o "**false**" podremos configurar nuestra API para que solo nos devuelva las ya hechas si enviamos "true" o viceversa si enviamos "false".

Utilizando estos parámetros, nuestra URL quedaría así:

```
https://api-tasks.cosasdedevs.com/v1/tasks?page=1&limit=10&is_done=true
```

Parámetros de ruta (path params)

Estos parámetros **forman parte de la ruta**, son obligatorios y pueden ser más difíciles de interpretar dentro de la URL a simple vista, ya que estos **no se asignan en un formato clave-valor**.

Ahora os voy a mostrar unos ejemplos de unos parámetros de ruta para que quede más claro:

```
https://cosasdedevs.com/posts/1/
```

```
https://cosasdedevs.com/posts/2/
https://cosasdedevs.com/posts/3/
```

Tenemos tres URLs que comparten esta parte de la ruta "/posts/" y en el **siguiente segmento tienen un número distinto**. En este caso podrían corresponder a un identificador único en mi base de datos y según el número de identificador enviado, mi web devolvería la información asociada a ese identificador.

Además de números, también podemos utilizar cadenas de texto como hago en mi blog:

```
https://cosasdedevs.com/posts/retornar-html-fastapi/
https://cosasdedevs.com/posts/capturar-excepciones-no-controladas-fastapi/
https://cosasdedevs.com/posts/fastapi-form-recibir-parametros-formulario/
```

En este caso, **cada cadena de texto en color verde funciona como clave única** para cada uno de mis posts y de esta forma puedo obtener el post asociado a esa clave.

También podemos enviar más de un parámetro por ruta separado por la barra inclinada "/". Al final todo depende de la configuración de nuestra web o API. Ejemplos:

```
https://cosasdedevs.com/posts/python/bucle-while-python/
https://cosasdedevs.com/posts/python/bucle-for-python/
https://cosasdedevs.com/posts/python/break-continue-pass-python/
```

En estos ejemplos habríamos creado como primer parámetro una categoría, en este caso Python, añadimos una barra inclinada y el siguiente parámetro sería el identificador único del post.

¿Y para qué son útiles este tipo de parámetros o por qué no usar parámetros de consulta en los que además podemos añadir una clave que mejora su legibilidad?

Por ejemplo, me podrías decir, que en vez de emplear:

```
https://cosasdedevs.com/posts/1/
```

Podríamos usar:

```
https://cosasdedevs.com/posts?id=1
```

Si y no. Lo primero porque **los parámetros de ruta se utilizan para identificar recursos** como puede ser el post al que corresponde el identificador 1. **Los parámetros de consulta son empleados para realizar filtros u ordenar información**.

También (esto tiene que ver con web, pero os lo dejo la información para que lo conozcáis) **en cuanto a optimización para posicionarte en un buscador** como Google, puedes tener problemas al intentar posicionar una URL con parámetros de consulta.

Otra de sus utilidades es a la hora de centralizar funcionalidades. Imagínate que en las URLs que pongo a continuación no estés utilizando los parámetros de ruta:

```
https://cosasdedevs.com/posts/retornar-html-fastapi/
https://cosasdedevs.com/posts/capturar-excepciones/
https://cosasdedevs.com/posts/recibir-parametros-form/
```

Tendrías que editar el proyecto cada vez que quisieras añadir una nueva ruta y además podríamos caer en malas prácticas como código duplicado.

¿Qué es el protocolo HTTP?

Sus siglas significan Hypertext Transfer Protocol (Protocolo de transferencia de hipertexto) y **este es el protocolo de comunicación que permite las transferencias de información en la World Wide Web** (red informática mundial).

Básicamente, es lo que **nos permite escribir en un navegador una URL** y el **servidor al que apunta transfiera la información de esa URL hasta nuestro navegador** para nosotros poder visualizar la información. Y aquí, cuando decimos navegador, también podemos hablar de otras herramientas que pueden funcionar como cliente HTTP (CURL, Postman, Imsomnia, etc.).

¿Y por qué nos interesa el funcionamiento de este protocolo? Pues bien, cuando trabajemos con una API, ya sea creándola o comunicándonos con ella, **será a través de este protocolo**. Por lo tanto, es de vital importancia entender su funcionamiento

Petición (Request) HTTP

Cuando nosotros accedemos a una página web desde el navegador o cliente HTTP, **estamos realizando una petición HTTP a esa web**.

El navegador o herramienta se comunicará con la web enviando un mensaje. **Este mensaje constará de cuatro partes que son el método, cabeceras, versión del protocolo y el cuerpo del mensaje (body)**.

A continuación voy a explicar de que trata cada parte y más adelante veremos como enviarlas en la petición.

Métodos HTTP

Los métodos HTTP **definen que tipo de acción queremos realizar**. Existen 9 tipos distintos, aunque nosotros vamos a ver los 5 más usados.

GET

El método GET se emplea en los casos en los que **queramos recibir datos**. En nuestro ejemplo utilizaremos el método GET para obtener el listado de nuestras tareas o la información de una tarea en concreto.

Otro ejemplo de petición GET y del que nunca somos conscientes es cuando accedemos a una página web desde el navegador.

POST

El método POST se utiliza cuando queremos **enviar información que podrá ser guardada o tratada por el servidor**. En nuestro ejemplo, emplearíamos el método POST para crear una nueva tarea o usuario.

PUT

El método PUT se usa cuando queremos **enviar información para actualizar todos los datos de un recurso**. Siguiendo nuestro ejemplo, este método lo emplearíamos cuando quisiéramos actualizar **TODA** la información de una tarea en concreto.

PATCH

El similar al método PUT, pero en este caso **solo queremos realizar una actualización parcial del recurso**. Siguiendo nuestro ejemplo, podríamos utilizar este método cuando únicamente queramos actualizar el estado de una tarea o su título.

DELETE

El método DELETE se emplea cuando **queremos eliminar un recurso en concreto**. Por ejemplo, eliminar una tarea.

Cabeceras (headers)

Las cabeceras **son parámetros que podemos enviar tanto en la petición como en la respuesta** y su función suele ser informativa.

Existen varios tipos de cabeceras que ya tienen un uso predeterminado y además podemos crear nuestras propias cabeceras para darles el uso que queramos. Al final se podría ver como otra forma de enviar parámetros.

Más adelante veremos en que casos necesitaremos usar alguna de ellas, pero quiero que os quedéis con el nombre de dos que son **Authorization** y **Content-type**, ya que se suelen utilizar cuando trabajamos con APIs.

La cabecera Authorization será enviada cuando necesitamos algún token de autenticación para trabajar con una API.

En nuestro ejemplo, al autenticarnos en nuestra API y obtener de ella un token de autenticación, este nos permitiría identificarnos como nuestro usuario y de esta manera, cuando generemos una tarea, esta estará relacionada con nuestro usuario. Podremos ver y trabajar solo con nuestras tareas y **no podremos acceder a las de los demás usuarios**.

La cabecera **Content-Type** se envía cuando usamos métodos como **POST**, **PUT** o **PATCH** en los que necesitamos mandar información.

Gracias a esta cabecera podremos indicar en que formato estamos enviando la información.

Si queréis conocer otras cabeceras HTTP os dejo el siguiente enlace:

https://es.wikipedia.org/wiki/Anexo:Cabeceras_HTTP

URL

La URL en la que realizar la petición.

Cuerpo del mensaje (body)

El cuerpo es opcional y lo utilizaremos cuando queramos **enviar información a una web o API**, como suele ser en los casos en los que usamos los **métodos POST, PUT y PATCH**.

Según el valor que le demos a la cabecera **Content-type**, el cuerpo del mensaje se enviará con un formato u otro.

El tema del formato puede sonar un poco confuso, así que para dejar más claro este punto, vamos a hablar de los **content-types** más empleados cuando enviamos información a una web o API.

Content-Type: application/x-www-form-urlencoded

Este caso es el más común. **La información se envía en pares clave-valor**. Una caso de uso en el que podemos ver esta cabecera es cuando se envía la información en un formulario HTML.

Content-Type: multipart/form-data

Esta cabecera es similar a la anterior, pero aparte de parámetros también nos **permitirá adjuntar archivos**.

Content-Type: application/json

Podemos utilizar esta cabecera cuando queramos enviar el mensaje en formato JSON. Recordad que el formato JSON es bastante flexible, por lo que nos dará mucha libertad a la hora de transmitir información.

También podemos usar otros **content-types** para enviar cadenas de **texto plano, XML, HTML**, etc.

Al final **la API o web es la que define en que formato quiere recibir la información**, así cuando estemos construyendo una API es importante documentar como queremos recibir la información para facilitar el trabajo a la persona que tenga que trabajar con nuestra API.

Respuesta HTTP

Siempre que **lanzamos una petición, obtendremos una respuesta por parte de la web o API**, al menos que el servicio por cualquier motivo no esté disponible.

Si ese es el caso, cuando se cumpla un tiempo de espera (definido desde el servidor) nos avisará de que no hemos podido comunicarnos correctamente.

Al igual que en la petición, la respuesta se compone de varias partes y aquí veremos las más relevantes que son el **código de estado**, **cabeceras** y la **respuesta**.

Código de estado (status code)

Los **códigos de estado son representados por un número y su labor es informativa**. Van desde el número 100 hasta el 599 y se dividen según su tipo de información en rangos de 100:

- Del 100 hasta 199 se encuentran las respuestas informativas.
- Del 200 hasta el 299 las respuestas en las que todo ha ido bien.
- Las redirecciones se encuentran desde el 300 hasta el 399.
- Los errores por parte del cliente van del 400 hasta el 499.
- Los errores por parte del servidor van desde el 500 hasta el 599.

A continuación voy a explicarte cuáles son los más comunes:

200 OK

Significa que la petición ha tenido éxito. En nuestra API de ejemplo, un caso puede ser que solicitemos un listado de tareas y las retorne correctamente.

201 CREATED

La petición ha tenido éxito y además se ha creado un nuevo recurso. Este caso podría ser cuando generamos una nueva tarea o usuario.

204 NO CONTENT

La petición ha tenido éxito, pero no se devuelve información. En algunos casos no necesitaremos devolver ninguna información. Para esas ocasiones existe este estado. Un ejemplo puede ser cuando eliminamos un recurso. Veremos un caso de uso más adelante.

301 MOVED PERMANENTLY

Este estado significa que **la URL a la que estamos realizando una petición ha sido modificada y ahora apunta a otra dirección**.

En el caso de mi blog, podría suceder que actualizase una ruta de un post en concreto para mejorar su posicionamiento en buscadores, pero tampoco quiero perder el tráfico que ya te consigue la ruta anterior.

Entonces debería realizar una configuración en mi servidor para que cuando apunten a la ruta original, este, redirija a la nueva.

307 INTERNAL REDIRECT

Significa que **nuestro servidor necesita efectuar una redirección interna**. Por ejemplo, que hayas hecho una configuración en tu servidor para que si intentan **acceder por HTTP a tu web, los redireccione a HTTPS**.

También te habrás fijado que puedes acceder a una página web añadiendo www o sin ello:

```
https://www.cosasdedevs.com
https://cosasdedevs.com
```

Por posicionamiento en buscadores, un sitio web también se debe configurar para que en uno de los dos casos redireccione al otro, ya que si no los buscadores lo interpretan como dos webs distintas y esto te puede perjudicar en el posicionamiento.

En el caso de una API no afectaría, pero está bien que lo tengas en cuenta para cuando trabajes en una web.

400 BAD REQUEST

Significa que la información enviada por parte del cliente no ha podido ser procesada. Por ejemplo, al crear un usuario, en la información que enviamos puede faltar algún dato.

401 UNAUTHORIZED

Significa que es necesario utilizar algún medio de autenticación para trabajar con el recurso. En nuestro ejemplo, intentar acceder a un listado de tareas sin enviar nuestro token de autenticación.

403 FORBIDDEN

No tenemos los permisos necesarios para trabajar con un recurso. Por ejemplo, intentar acceder a la tarea de otro usuario.

404 NOT FOUND

Significa que no encuentra lo que estamos buscando. Por ejemplo, intentar obtener una tarea que no existe.

405 METHOD NOT ALLOWED

Significa que la **ruta** a la que estamos haciendo **la petición no permite el método solicitado**, por ejemplo una ruta que solo funcione con el método GET, al realizar otro método devolvería este estado.

422 UNPROCESSABLE ENTITY

Significa que la petición tiene el formato correcto , pero no puede ejecutar el proceso. Por ejemplo, al intentar crear un usuario, si enviamos un email con un formato inválido.

429 TOO MANY REQUESTS

Significa que se han hecho demasiadas peticiones al dominio en general o a una ruta en concreto. Este estado se devuelve en los casos en los que una API tiene un límite de peticiones en un tiempo.

Por ejemplo, se podría poner un límite de 60 peticiones en 10 minutos y superado ese límite devolver este estado y un mensaje de error.

500 INTERNAL SERVER ERROR

Significa que ha ocurrido un error, pero el servidor no puede dar más información. Este caso podría aparecer, por ejemplo, en una excepción no controlada.

Más adelante veremos unos cuantos consejos sobre que hacer cuando estamos construyendo una API y ocurre este error.

503 SERVICE UNAVAILABLE

Significa que por alguna razón el servidor no puede dar la respuesta solicitada al usuario, ya sea porque esté sobrecargado, en mantenimiento o se haya roto nuestra API.

504 GATEWAY TIMEOUT

Significa que se ha superado el tiempo máximo en el que se puede responder a una petición. Por ejemplo, si la solicitud realiza una consulta a la base de datos que supera el tiempo de espera máximo definido en el servidor.

Que más debemos saber acerca de los estados HTTP

Los códigos de estado mostrados anteriormente serían los que considero más importantes y los que deberías conocer.

También y antes de seguir quiero comentarte algo muy relevante y es que **los códigos de estado NO SIEMPRE pueden indicar realmente que está pasando**.

Al final, el equipo o persona que desarrolla la API es el que indica que código de estado mostrar, por lo que **un mal desarrollo podría hacer que retornase un código de estado que no se corresponde con la realidad**.

Por ejemplo, esta misma semana he estado trabajando con una API en la que si ocurre una excepción que no está controlada, esta devuelve toda la traza de error en texto plano (se espera un JSON) y además retorna el estado 200. Nada que ver con la realidad, así que os aconsejo que estéis atentos y no deis nada por hecho.

Si quieres conocer otros códigos de estado te dejo este enlace: https://developer.mozilla.org/es/docs/Web/HTTP/Status

Cabeceras (headers)

Al igual que en la petición, la respuesta devuelve cabeceras. También son parámetros con información, esta vez para el cliente. Nos pueden indicar cosas como el tamaño de la respuesta, en que formato viene, cookies generadas, etc. Ya que ya las vimos en el apartado de peticiones, HTTP no me voy a explayar más en este tema.

Respuesta (response)

La respuesta es la información que el servidor devuelve al usuario. Cuando accedemos a una web mediante el navegador, la respuesta sería el código HTML que el navegador interpreta y construye para que podamos visualizarlo.

En el ejemplo de nuestra API sería el contenido según la petición que realizásemos. Si, por ejemplo, solicitamos nuestro listado de tareas, devolverá un JSON con la información.

Además, gracias a la cabecera de la respuesta **content-type** podremos saber el formato en el que viene la respuesta (HTML, JSON, XML, texto plano, etc.).

Arquitectura REST

El término REST, que significa **REpresentational State Transfer**, es una arquitectura que funciona con el protocolo HTTP y tiene unos puntos de diseño claves que son:

Un protocolo cliente/servidor sin estado

Esto quiere decir que cada petición HTTP tiene toda la información necesaria para su procesamiento. Por lo tanto, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre ellos, como podrían ser cookies o algún sistema para mantener la sesión de un usuario.

Como recordaréis en el ejemplo de nuestra API, **al autenticarnos recibimos un token** que usaremos para poder trabajar con nuestras tareas, pero en ningún momento vamos a guardar una sesión de usuario ni nada parecido.

Conjunto de operaciones bien definidas

Este punto se refiere a que podemos hacer con la API. En nuestro ejemplo, pues operaciones como crear una tarea, poder recuperar su información o la información de todas nuestras tareas, actualizarlas y borrarlas.

Uso de hipermedios

Esto quiere decir que la API debe poder **ayudarte a navegar entre distintos recursos**.

Por ejemplo, si recordáis el punto donde explicamos los parámetros de consulta, las tareas estaban paginadas, por lo tanto, debíamos

hacer varias peticiones para obtener el listado de tareas de un usuario si estas, superaban un límite.

Pues bien, en este caso, para ayudar al usuario, podríamos añadir a la respuesta dos links. **Uno con la URL para obtener las tareas de la página anterior y otro a la página siguiente**. Si no existen, podemos enviarlos vacíos. De esta manera, el usuario podría navegar por todo su listado de tareas hasta que el link a la página siguiente aparezca vacío.

Si no ha quedado muy claro, no te preocupes, esto lo veremos en detalle cuando construyamos nuestra API.

Sintaxis universal para identificar recursos

Significa que cada ruta de nuestra API **identifica a un recurso en concreto**. Si por ejemplo realizamos una petición para obtener la tarea con el identificador 1, la API retornará la información de la tarea con ese identificador.

Formato de construcción de las rutas

A la hora de construir una ruta, en **REST se deben nombrar los recursos como sustantivos**, por ejemplo, usuarios, tareas, facturas, etc. Gracias **al método HTTP que enviemos, indicamos el tipo de acción a ejecutar**. Estas rutas son conocidas como URI.

URI

Significa **identificador único del recurso** y son rutas que en sus nombres **NO DEBEN** especificar ninguna acción, ya que como he indicado antes, **las acciones las podremos especificar mediante el método HTTP**.

Vamos a verlo en un ejemplo en el que tendríamos que definir una URI para generar una tarea.

Grupo 1:

```
https://ejemplo.com/tareas/crear ❌
https://ejemplo.com/tareas/borrar ❌
https://ejemplo.com/tareas/actualizar ❌
https://ejemplo.com/tareas/todas ❌
https://ejemplo.com/tareas/id/1 ❌
```

Grupo 2:

```
https://ejemplo.com/tareas/ ✅
https://ejemplo.com/tareas/1 ✅
```

Si os fijáis en el primer grupo de **URLs** estamos indicando la acción que queremos hacer en la URI. **Esto es lo que debemos evitar,** porque como ya hemos comentado, lo indicamos gracias al método HTTP.

Sin embargo, en el segundo grupo de **URLs** solo estamos indicando la entidad o sustantivo que sería la forma correcta, ¿Qué queremos realizar la acción de crear? Perfecto, utilizamos el método **POST**, después le indicamos a nuestra API que cuando reciba una petición a esa ruta con el método **POST** lo que queremos hacer es crear una tarea y listo.

En el caso de actualizar o borrar, también podemos usar la segunda URL (en la que enviamos el identificador de tarea por parámetro) y emplear los métodos HTTP PUT, PATCH o DELETE.

Si queremos obtener la información de una tarea en concreto, podemos añadir como parámetro de ruta su identificador (como se puede ver en la segunda URL). Ejecutamos una petición de tipo GET y listo.

Por último, para obtener todo el listado de tareas, podemos usar la primera URL con el método GET.

Como podéis ver, **con únicamente dos URLs podemos cubrir todas la acciones que necesitemos**.

No os preocupéis si no veis muy claro ahora mismo estos conceptos, porque **vamos a ver como construir nuestra API con la arquitectura REST** gracias a lo todo lo que hemos visto hasta ahora.

Nombrar los recursos en plural o singular

En cuanto a si el recurso debe escribirse en **plural o singular**, estuve leyendo este debate en Stack Overflow: https://stackoverflow.com/questions/6845772/should-i-use-singular-or-plural-name-convention-for-rest-resources

En uno de los comentarios hacían referencia a una guía de Google que también os dejo aquí: https://cloud.google.com/apis/design/resource_names#collection_id

En ella indica que **se deben crear en plural y con formato lowerCase**. Solo deberíamos usar el singular en el caso en el que no exista una forma adecuada en plural.

Jerarquías

Cuando accedamos a más de un nivel de recursos debemos mantener una orden jerárquico en la URI. Ejemplo para acceder a una tarea de un usuario:

```
https://ejemplo.com/tareas/5/usuarios/1 ❌
https://ejemplo.com/usuarios/1/tareas/5 ✅
```

En la jerarquía de esta API, las tareas dependen totalmente del usuario, por lo que el recurso principal sería el usuario y después indicaríamos la tarea que queremos obtener de este usuario.

Utilizar la arquitectura REST con lo que hemos aprendido

Ahora que conocemos un poco mejor la **arquitectura REST,** vamos a aprender como construir nuestro ejemplo de API con el que hemos trabajado durante toda la guía.

Antes de empezar

Para verlo todo más claro, he construido una API en la que tu mismo podrás probar desde un **cliente HTTP** todos los endpoints que creemos aquí.

Para realizar las peticiones a nuestra API, vamos a utilizar el **cliente CURL**. No te preocupes si no conoces CURL, ya que os indicaré el comando completo para cada petición.

CURL

CURL es una herramienta de línea de comandos que permite realizar peticiones HTTP. Suele estar instalada en todas las

terminales de Linux, aunque también se puede instalar en la terminal de Windows y Mac.

También te dejo esta página donde puedes añadir un comando de CURL y esta la ejecutará, de esta forma podrás probarlo tu mismo si no tienes CURL o no quieres instalarlo.
https://reqbin.com/curl

Por último, os dejo un ejemplo de comando CURL para explicaros su funcionamiento:

```
curl -X POST 'https://api-tasks.cosasdedevs.com/v1/tasks' \
-H 'Authorization: xxxxx' \
-H 'Content-Type: application/json' \
-d '{
    "title": "Mi primera tarea"
}'
```

Primero debemos escribir el comando "**curl**". Después solo debemos añadir las flags (banderas) que necesitemos.

Para indicar el método que queremos utilizar, escribimos "**-X**", luego el método que en este caso es POST y la URL en la que vamos a realizar la petición.

Para añadir cabeceras, empleamos "**-H**" por cada cabecera que queramos añadir y después entre comillas, la cabecera en **formato clave: valor**.

Por último, si necesitamos enviar información, usamos "**-d**" y después entre comillas la información a enviar.

API de prueba

La API de prueba está basada en el ejemplo con el que hemos trabajado durante toda la guía. Obviamente, simulará todas la acciones de creación, borrado y actualización, pero **NO persistirán los datos**. Para realizar las pruebas he creado tres usuarios, el cual uno es administrador y cada uno tiene asignadas 50 tareas.

Los usuarios existentes y lo cuales podrán realizar login son:

admin@cosasdedevs.com
user2@cosasdedevs.com
user3@cosasdedevs.com

Los tres tienen la misma contraseña que es “admin123”. El primero es el usuario que tiene permisos de administración y el que podrá utilizar endpoints especiales.

El proyecto está construido en PHP. Si quieres echarle un vistazo, te dejo su repositorio:
https://github.com/albertorc87/easyapi_tasks_api

Por último, la URL de nuestra API es:
https://api-tasks.cosasdedevs.com/v1/

Como puedes observar, en el primer segmento indicamos la versión de la API (v1), y a partir de ahí, generaremos todos nuestros recursos. De como versionar nuestra API, hablaremos más adelante. Por ahora solo quédate con que la URL base en la cual crearemos todos nuestros endpoints será la indicada anteriormente.

Ruta de registro y autenticación

Para poder crear tareas, lo primero que necesitaremos es estar autenticado y para poder autenticarnos necesitaremos crear un endpoint donde poder registrarnos, así que vamos a ello.

Registro

Como hemos dicho antes, utilizaremos un sustantivo y, ya que se trata de usuarios, vamos a **definir la ruta "users"**:

```
https://api-tasks.cosasdedevs.com/v1/users
```

Puesto que la acción a realizar es un nuevo registro, **el método HTTP a utilizar será de tipo POST**.

Como datos para el registro de un usuario vamos a definir que tendrá un email, nombre de usuario y contraseña.

Ahora viene una parte muy importante y es que según como definamos la forma de recibir los datos, **tendremos que elegir una cabecera u otra** y eso también implicará cambios en cómo enviaremos los datos en el cuerpo del mensaje.

Si la API la hemos construido de tal forma que esta ruta reciba los datos como un formulario, debemos enviar la siguiente cabecera en la petición:

```
Content-Type: application/x-www-form-urlencoded
```

El cuerpo del mensaje que es donde enviaríamos los datos sería este:

```
email=user4@cosasdedevs.com&username=user4&password=strongpass
```

Como veis es similar a enviar la información como parámetros de consulta (query) pero sin añadir la interrogación inicial.

En el caso de utilizar otro formato como por ejemplo JSON, usaríamos la siguiente cabecera:

```
Content-Type: application/json
```

El cuerpo del mensaje obviamente debería tener el formato JSON:

```
{
  "email": "user4@cosasdedevs.com",
  "username": "user4",
  "password": "admin123"
}
```

Si decidimos añadir la opción de poder subir un avatar, aquí cambia un poco la historia, deberíamos utilizar la siguiente cabecera:

```
Content-Type: multipart/form-data
```

Como comenté cuando expliqué el funcionamiento del envío del cuerpo del mensaje, esta cabecera es exactamente igual a **application/x-www-form-urlencoded**, pero **nos permite enviar archivos en nuestras peticiones**.

Para la creación de un usuario, finalmente he decidido que voy a enviar la información como un formulario, así que utilizaremos la cabecera correspondiente.

La **API al recibir la petición procesará la información** y aquí debemos controlar unas cuantas cosas como la validación de datos.

Unos ejemplos de validación pueden ser que el email tenga un formato válido, comprobar si el email o usuario ya existen o definir un número mínimo de caracteres para la contraseña.

En este caso **el estado HTTP que podemos dar es el 422 (unprocessable entity)**, ya que por un error en la validación no se ha podido procesar la petición. Como respuesta podemos devolver un JSON con un mensaje de error:

```
{
  "status": "error",
  "error": "Email already exists"
}
```

También importante, **añadir en todas nuestras respuestas la cabecera indicando el formato** de esta que en este caso y en todas las respuestas de la API ya definimos que sería **en formato JSON**:

```
Content-Type: application/json
```

Si todo ha ido bien, **podemos devolver el estado HTTP 201 (created)**, ya que estamos creando un nuevo registro y como respuesta podemos retornar la información del usuario recién creado:

```
{
   "status": "success",
   "data": {
      "id": 4,
      "username": "user4",
      "email": "user4@cosasdedevs.com"
   }
}
```

Crear el usuario con CURL

```
curl -X POST 'https://api-tasks.cosasdedevs.com/v1/users' \
-H 'Content-Type: application/x-www-form-urlencoded' \
-d 'username=user4&email=user4@cosasdedevs.com&password=admin123'
```

Login

Una vez nos hemos registrado procederemos a autenticarnos para poder **usar todas las herramientas de nuestra API**. Para autenticarnos vamos a crear el siguiente endpoint:

```
https://api-tasks.cosasdedevs.com/v1/login
```

El método que utilizaremos será de tipo **POST, ya que vamos a enviar información a la API**.

Como **content-type** en la cabecera utilizaremos el valor para formularios:

```
Content-Type: application/x-www-form-urlencoded
```

El cuerpo del mensaje que es donde enviaríamos los datos sería este:

```
email=admin@cosasdedevs.com&password=admin123
```

Si todo ha ido bien, **devolveremos un estado HTTP 200 y el token de autenticación** en formato JSON. Por ejemplo así:

```
{
   "status": "success",
   "data": {
      "token": "xxxxxx"
   }
}
```

Sin embargo, si introduce mal el usuario o contraseña **podemos devolver el estado HTTP 401 (unauthorized)** y un mensaje de error.

```
{
   "status": "error",
   "error": "Invalid user or password"
}
```

Autenticación con CURL

```
curl -X POST 'https://api-tasks.cosasdedevs.com/v1/login' \
-H 'Content-Type: application/x-www-form-urlencoded' \
```

```
-d 'email=admin@cosasdedevs.com&password=admin123'
```

Si ejecutáis este comando, guardaros el token para así poder probarlo con las demás peticiones.

Tareas

Ahora que ya tenemos un usuario, podemos empezar con el sistema de tareas. Para trabajar con las tareas, vamos a realizar lo que sé conoce como un **CRUD, que significa Create Read Update Delete**, o sea, las acciones de crear, listar, actualizar y borrar.

Crear una tarea

Lo primero que vamos a hacer es crear una tarea. De una tarea vamos a querer guardar un título, la fecha de creación, el estado (si está hecha o no) y un identificador.

De estos cuatro valores solo necesitaremos enviar el título de la tarea, puesto que la fecha de creación se añadirá al insertarla junto con la fecha actual. El estado lo guardaremos como no hecho por defecto y el identificador se generará automáticamente.

El primer punto es definir el endpoint que en este caso lo vamos a llamar “tasks”.

```
https://api-tasks.cosasdedevs.com/v1/tasks
```

Como vamos a enviar información la petición que haremos será de tipo POST y en este caso podemos enviar la información en formato JSON. Para ello, recordad que debemos utilizar la cabecera para contenido JSON y, ya que este endpoint necesita autenticación,

también tendremos que pasar el token que generamos anteriormente:

```
Content-Type: application/json
Authorization: strongtoken
```

Y en el cuerpo del mensaje enviaremos el título.

```
{
        "title": "Mi primera tarea"
}
```

Si todo ha ido bien, **devolveremos el código de estado HTTP 201** y como respuesta la tarea recién creada.

```
{
  "status": "success",
  "data": {
    "task": {
      "id": 1,
      "user_id": 1,
      "title": "Mi primera tarea",
      "created_at": "2022-04-30 18:56:43",
      "updated_at": null,
      "is_done": false
    },
    "msg": "Task has been created successfully"
  }
}
```

Si hay algún fallo de validación (título vacío, superar un límite de caracteres o quedarnos cortos, etc.), **podemos devolver el 422 (unprocessable entity)** y un mensaje de error.

```
{
   "status": "error",
   "error": "The title must have at least 5 characters"
}
```

También puede que haya **caducado nuestro token o no sea correcto**, por lo que podemos enviar **el estado 401** y el mensaje de error:

```
{
   "status": "error",
   "error": "Your token has expired, please login again"
}
```

A partir de ahora creo que puedes hacerte una idea de como funcionan los errores, así que para los siguientes endpoints solo voy a incluirlos si aparece algún tipo de error que no hayamos contemplado hasta ahora.

Creación de una tarea con CURL

```
curl -X POST 'https://api-tasks.cosasdedevs.com/v1/tasks' \
-H 'Authorization: xxxxx' \
-H 'Content-Type: application/json' \
-d '{
   "title": "Mi primera tarea"
}'
```

Si lo vais a probar, recordad sustituir en el valor de la cabecera "Authorization", xxxxx por vuestro token.

Obtener el listado de tareas del usuario

Ahora que ya podemos crear tareas, **es hora de listar todas las tareas de nuestro usuario**. Para ello utilizaremos el mismo endpoint que para la creación, pero en este caso el método que usaremos será **GET** y recordad que también debemos **enviar el token de autenticación**.

```
https://api-tasks.cosasdedevs.com/v1/tasks
```

Devolveremos código de estado HTTP 200 (ok) y la respuesta será el listado de tareas del usuario:


```
{
  "status": "success",
  "data": {
    "page": 1,
    "limit": 10,
    "filters": {
      "is_done": null
    },
    "pagination": {
      "total": 50,
      "tasks_per_page": 10,
      "previous": "",
      "next": "https://api-tasks.cosasdedevs.com/v1/tasks?page=2&limit=10"
    },
    "tasks": [
      {
        "id": 1,
        "user_id": 1,
        "title": "1 task",
        "created_at": "2022-04-28 16:05:08",
        "updated_at": "2022-04-28 16:05:08",
```

```
            "is_done": true
        },
        {
            "id": 2,
            "user_id": 1,
            "title": "2 task",
            "created_at": "2022-04-28 16:05:08",
            "updated_at": "2022-04-28 16:05:08",
            "is_done": true
        },
        {
            "id": 3,
            "user_id": 1,
            "title": "3 task",
            "created_at": "2022-04-28 16:05:08",
            "updated_at": "2022-04-28 16:05:08",
            "is_done": true
        }...
    }
  }
}
```


Como podéis observar en la respuesta, aquí **hemos añadido todo el sistema de paginación, hipermedios** y además podemos usar el filtro "**is_done**" por si solo queremos obtener las tareas realizadas o pendientes de hacer.

Por ejemplo, para obtener las tareas de la página 2 y que el estado sea realizada, la URL sería esta:

```
https://api-tasks.cosasdedevs.com/v1/tasks?page=2&limit=10&is_done=true
```

Obtener la lista de tareas con CURL

```
curl -X GET 'https://api-tasks.cosasdedevs.com/v1/tasks' \
-H 'Authorization: xxxxx'
```

Obtener una tarea en concreto

Cuando queramos obtener una tarea en concreto, utilizaremos **el método GET** y en este caso efectuaremos un pequeño cambio en la URL que **es añadir el identificador como parámetro de ruta**.

```
https://api-tasks.cosasdedevs.com/v1/tasks/30
```

Nuestra API recuperará el identificador y de esta forma sabrá que tarea debe retornar.

El código de estado será el 200 (ok) y esta sería la respuesta

```
{
   "status": "success",
   "data": {
      "id": 30,
      "user_id": 1,
      "title": "30 task",
      "created_at": "2022-04-28 16:05:08",
      "updated_at": null,
      "is_done": false
   }
}
```

Si por ejemplo no existe una tarea con el identificador enviado, **podemos devolver el estado 404 (not found)** para decir que no hemos encontrado la tarea y un mensaje de error.

Si el identificador enviado es de una tarea que **no pertenece al usuario,** podemos **devolver el estado 403 (forbidden)** aunque **por**

seguridad podríamos **devolver el estado 404 (not found)** para evitar que un usuario ajeno a esa tarea ni siquiera tenga la opción de saber que existe.

```
{
   "status": "error",
   "error": "Task not found"
}
```

Obtener una tarea con CURL

```
curl -X GET 'https://api-tasks.cosasdedevs.com/v1/tasks/30' \
-H 'Authorization: xxxxx'
```

Actualizar una tarea

Para actualizar una tarea utilizaremos el mismo endpoint que usamos para obtener una tarea, es decir, este:

```
https://api-tasks.cosasdedevs.com/v1/tasks/30
```

En el que **enviamos el identificador como parámetro de ruta**.

Cómo método **vamos a utilizar PUT, aunque también valdría PATCH**, ya que podemos realizar una actualización parcial o completa de los datos que son editables (en este caso, “title” y “is_done”).

Como cabeceras recordad que tenemos que enviar el token y en cuerpo del mensaje podremos actualizar el título y si está realizada:

```
{
   "title": "30 task update",
    "is_done": true
}
```

Actualizar una tarea con CURL

```
curl -X PUT 'https://api-tasks.cosasdedevs.com/v1/tasks/30' \
-H 'Authorization: xxxxx' \
-H 'Content-Type: application/json' \
-d '{
   "title": "30 task update",
    "is_done": true
}'
```

Borrar una tarea

Por último vamos a ver como borrar una tarea. Al igual que en el caso anterior, emplearemos **el endpoint en el que pasamos el identificador como parámetro de ruta,** pero en este caso usaremos el método **DELETE**.

```
https://api-tasks.cosasdedevs.com/v1/tasks/30
```

Gracias al identificador, nuestra API sabrá que tarea debe borrar.

Como respuestas **podemos dar el estado 200 (ok)** y devolver un mensaje indicando que se ha borrado de forma satisfactoria o también podríamos **devolver el estado 204 (no content)** y la respuesta vacía.

Borrar una tarea con CURL

```
curl -X DELETE 'https://api-tasks.cosasdedevs.com/v1/tasks/30' \
-H 'Authorization: xxxxx'
```

Si ejecutas el comando, en nuestro caso estaremos dando una respuesta vacía y el estado HTTP 204.

Administración

Como comenté al principio de esta sección, **disponemos de un usuario con privilegios de administración** (el usuario con email: admin@cosasdedevs.com). En nuestra API, si tienes esos privilegios podrás acceder a tres endpoints adicionales que explicaré a continuación.

Obtener el listado de usuarios existentes

Con todo lo aprendido hasta ahora, este endpoint no tiene mucho misterio. Utilizamos el nombre de recurso "**users**" que ya creamos anteriormente para crear un usuario, pero esta vez el método a usar será GET.

```
https://api-tasks.cosasdedevs.com/v1/users
```

La respuesta será los tres usuarios existentes:

```
{
  "status": "success",
  "data": [
    {
      "id": 1,
      "email": "admin@cosasdedevs.com",
      "username": "admin",
      "is_admin": true,
```

```
        "is_active": true
      },
      {
        "id": 2,
        "email": "user2@cosasdedevs.com",
        "username": "user2",
        "is_admin": false,
        "is_active": true
      },
      {
        "id": 3,
        "email": "user3@cosasdedevs.com",
        "username": "user3",
        "is_admin": false,
        "is_active": true
      }
    ]
}
```

En el caso de emplear un usuario que no tenga permisos de administración, **retornaremos un código de estado 403 (forbidden)** y la siguiente respuesta:

```
{
  "status": "error",
  "error": "You don't have the require permissions"
}
```

Obtener el listado de usuarios con CURL

```
curl -X GET 'https://api-tasks.cosasdedevs.com/v1/users' \
-H 'Authorization: xxxxx'
```

Poder ver el listado de tareas de un usuario

El anterior no tenía mucho misterio, pero este es más interesante, ya que aquí vamos a ver las jerarquías. Si por ejemplo queremos obtener las tareas del usuario con el identificador 3, nuestra URI debería ser la siguiente:

```
https://api-tasks.cosasdedevs.com/v1/users/3/tasks
```

Como comenté cuando hablé de las jerarquías, primero añadimos el recurso de usuarios, después su identificador y por último el recurso tasks.

La respuesta no la voy a añadir, ya que el formato es idéntico a cuando obtenemos las tareas de nuestro usuario.

Obtener la tarea de un usuario con CURL

```
curl -X GET 'https://api-tasks.cosasdedevs.com/v1/users/3/tasks' \
-H 'Authorization: xxxxx'
```

Poder ver una tarea en concreto de un usuario

En el caso de querer ver una tarea en concreto de un usuario, podemos utilizar la URI anterior, añadiendo al final el identificador de la tarea que estamos buscando:

```
https://api-tasks.cosasdedevs.com/v1/users/3/tasks/101
```

En este caso, si la tarea no pertenece al usuario, podemos devolver el aviso al administrador de que no pertenece en vez de mostrar el **error 404 (not found)**, ya que en este caso, al ser el administrador, no tenemos que ocultar si existe o no.

Unos consejos más a la hora de construir nuestra API

¡Listo! Ya tenemos definida nuestra **API** y ahora solo faltaría convertirlo en código, pero ese ya es tu trabajo ;).

Por último, voy a dejaros unos cuantos consejos más que seguro, os ayudarán a llevar vuestras APIs al siguiente nivel.

Utilizar un estándar para las respuestas.

Aquí quiero decir que siempre uses **un mismo formato de repuesta cuando retornes información al usuario**. Por ejemplo, podemos definir un JSON con dos claves que serán status y data.

Si todo ha ido bien podemos devolver el status con un valor de "**success**" y si hay algún error con el valor "**error**". En data, si todo ha ido bien podemos definir la respuesta para el usuario y si tenemos errores los mensajes de error.

Ejemplo de respuesta válida

```
{
	"status": "success",
	"data": {
		"title": "Mi primera tarea",
		"is_done": true
	}
}
```

Ejemplo de respuesta errónea

```
{
	"status": "error",
	"error": "El título es requerido"
```

```
}
```

¿Por qué hacer esto?

De esta forma le devolvemos **siempre al usuario un mismo formato de respuesta** y así le será más fácil trabajar con tu API.

¿Cómo montar esto en nuestra API?

Puedes **crear una clase que se encargue de realizar todas las respuestas de tu API** y ahí definir un método que reciba si todo ha ido bien o no y los datos. Después solo deberá retornar la información con el formato que definamos.

Te dejo un estándar que suelo utilizar cuando tengo que crear una API desde 0:
https://github.com/omniti-labs/jsend

Centralizar el sistema para que todos los errores no capturados pasen por el mismo lado

Cuando creas o trabajas en el mantenimiento de una API, es posible que se te **escape algún error que genere una excepción no controlada**. Hoy en día los frameworks disponen de funcionalidades para que puedas hacer pasar una excepción no controlada por una función y ahí decidir como tratarla.

Aquí **puedes enviarte un email, notificación de Slack o lo que sea con la traza del error** para poder revisarlo y corregirlo posteriormente. Al usuario le puedes enviar un mensaje genérico. Ejemplo:

```
{
```

```
    "status": "error",
    "error": "An error has occurred"
}
```

Si no estamos utilizando ningún framework, la mejor forma sería **centralizar todas las peticiones y respuestas y tener capturado todo en un bloque try catch con la excepción general**.

¿Por qué hacer esto?

Primero **por seguridad**. Si tenemos una excepción con una consulta SQL, podemos mostrar información comprometida de la traza del error al usuario. También podría ver que librerías utilizamos o el lenguaje de programación y una persona malintencionada podría aprovecharse de cualquier vulnerabilidad.

Adicionalmente, vas a quedar mejor de cara al usuario **si le muestras un error en el formato que él espera,** que si le llega una cadena de texto que puede que ni entienda.

Por último y también muy importante. Si lo haces como te he comentado anteriormente, podrás **capturar los errores y decidir como gestionarlos (guardar un log, enviarte un email, etc.) para solucionarlos posteriormente**.

En mi blog, he escrito un tutorial en el que explico cómo construir un framework con PHP para que veas todo el proceso. Te dejo el enlace por si le quieres echar un vistazo.
https://cosasdedevs.com/posts/construir-un-framework-sencillo-php/

Separar la API por versiones

Según vaya evolucionando y creciendo tu API, es posible que tengan que **convivir varias versiones de una misma API**.

¿Por qué ocurre esto?

A veces tendrás que realizar grandes cambios en toda la API, que pueden hacer que los endpoints que antes funcionaban de una manera, ahora se comporte de otra. Por ejemplo, que ahora cambien los nombres de los campos a enviar, se añadan nuevos o se eliminen.

Si esto sucede y un usuario que use tu API, ya sea por tiempo o porque no se entere no efectúa los cambios pertinentes, podrá tener problemas para comunicarse con tu API.

Para solucionar esto, en vez de editar tus endpoints ya existentes (repito, en cambios grandes que afecten a su comportamiento), **puedes generar una nueva versión de tu API con los nuevos cambios mientras mantienes por un tiempo la versión antigua**.

De esta forma le das al usuario un tiempo para que migre su configuración a la nueva versión sin dejarle “con el culo al aire”.

¿Cómo hacerlo?

Si vas a seguir trabajando dentro del mismo proyecto, puedes separar las distintas versiones en carpetas.

Por ejemplo, la primera versión en una carpeta llamada **v1**, la siguiente en **v2** y así sucesivamente.

En el caso de las URLs puedes utilizar también una nomenclatura similar.

Versión 1 de tu API:

```
https://api-tasks.cosasdedevs.com/v1/tasks
```

Versión 2 de tu API:

```
https://api-tasks.cosasdedevs.com/v2/tasks
```

Como puedes observar, **en la ruta indicamos la versión** y ya en la API según la versión que se reciba, emplearás los endpoints de una versión u otra.

Existen otras maneras de indicar la versión como podría ser enviando la versión por cabeceras o por parámetros de consulta, pero yo te recomiendo que emplees esta.

Si para una nueva versión de tu API creas un nuevo proyecto, ya deberás realizar configuraciones en tu servidor si vas a seguir utilizando el mismo dominio.

Esto ha sido todo

Gracias, de corazón, por leer mi guía para aprender a trabajar con APIs.

Espero que te ayude en tu camino y que hayas aprendido algo nuevo. Esta guía la he escrito para poder aportar mi granito de arena en esta comunidad con la que gracias a toda la información gratuita que comparte tanta gente he podido llegar a donde estoy y crecer como desarrollador, de forma que si no hubiera sido así esto habría sido imposible.

Te invito a que si te ha ayudado o quieres darme feedback, me escribas un MD por Twitter que estaré encantado de leerte :).

Sobre el autor

Llevo trabajando como programador desde hace más de 10 años. Me encontré con JavaScript cuando estaba estudiando y desde entonces no he podido parar.

Actualmente, trabajo como **Backend Developer** en el área de producto, sobre todo en el mantenimiento y desarrollo de nuevas Web APIs.

También he hecho mis pinitos en el front y la mayor parte de mi carrera ha sido Fullstack, pero mi corazón está en el Backend.

Puedes seguirme en Twitter y te dejo mi sitio web donde escribo tutoriales sobre **PHP, Python, Laravel, Django y FastAPI**.

Twitter: https://twitter.com/backalber
Mi sitio web: https://cosasdedevs.com

Fuentes de referencias

Para escribir esta guía me he valido de las siguientes referencias:

- Códigos de estado de respuesta HTTP
- Cabeceras HTTP
- Protocolo HTTP
- Arquitectura REST
- Cómo nombrar los recursos
- Artículo de REST en la Wikipedia
- Diferencia entre Web Service y API

Notas de la versión

Versión 1.1

- Uso de hipermedios.
- Singular o plural a la hora de nombrar nuestros recursos.
- Jerarquía en la URI.
- Explicar el funcionamiento del cliente CURL.
- Actualización de la construcción de nuestra API para utilizar una API real.
- Página del autor.

Versión 1.2

- Redefinición del concepto de API.
- Ejemplo de API que no es un Web Service.
- Actualización de links de Twitter.