

DWC

(Desarrollo Web en entorno cliente)



JavaScript

Tema 7.1

API REST

Índice

1.- ¿Que es una API?.....	1
2.- La arquitectura REST.....	1
3.- Llamadas a la API REST.....	3
4.- Clientes REST (Postman).....	4
5.- Json-Server.....	8
6.- Json-Server y Postman.....	11

1.- ¿Que es una API?

Se conoce como API (del inglés: «Application Programming Interface»), o Interfaz de programación de Aplicaciones **al conjunto de rutinas, funciones y procedimientos (métodos) que permite utilizar recursos de un software por otro, sirviendo como una capa de abstracción o intermediario.**

API REST es el sucesor de métodos anteriores como **SOAP** y **WSDL** cuya implementación y uso son un poco más complejos y requieren mayores recursos y especificaciones al ser usados.

2.- La arquitectura REST

La arquitectura REST (del inglés: Representational State Transfer) trabaja sobre **el protocolo HTTP**. Por consiguiente, los procedimientos o métodos de comunicación son los mismos que HTTP, siendo los principales: **GET, POST, PUT, PATCH y DELETE**.

Otros métodos que se utilizan en REST API son **OPTIONS** y **HEAD**. Este último se emplea para pasar parámetros de validación, autorización y tipo de procesamiento, entre otras funciones.

Otro componente de un REST API es el «**HTTP Status Code**», que le informa al cliente o consumidor del API que debe hacer con la respuesta recibida. Estos son una referencia universal de resultado, es decir, al momento de diseñar un RESTful API toma en cuenta utilizar el «Status Code» de forma correcta.

Por ejemplo, el código 200 significa OK, que la consulta ha recibido respuesta desde el servidor o proveedor del API.

Los códigos de estado más utilizados son:

- 200 OK
- 201 Created
- 304 Not Modified
- 400 Bad Request
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found
- 422 Unprocessable Entity
- 500 Internal Server Error (Error Interno de Servidor)

Una respuesta de un RESTFul API sería lo siguiente:

```
Status Code: 200 OK
Access-Control-Allow-Methods: PUT, GET, POST, DELETE, OPTIONS
Connection: Keep-Alive
Content-Length: 186
Content-Type: application/json
Date: Mon, 24 May 2016 15:15:24 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.4.9 (Win64) PHP/5.5.12
X-Powered-By: PHP/5.5.12
access-control-allow-credentials: true
```

Por lo general y mejor práctica, el cuerpo (Body) de la respuesta de un API es **una estructura en formato JSON**. Aunque también puede ser una estructura XML o cualquier otra estructura de datos de intercambio, incluso una personalizada. Sin embargo, como el objetivo es permitir que cualquier cliente pueda consumir el servicio de un API, lo ideal es mantener una estructura estándar, **por lo que JSON es la mejor opción**.

REST se ha convertido probablemente en la tecnología más utilizada hoy en día para el desarrollo de aplicaciones Web Cliente/Servidor en las que haya que solicitar o mandar al servidor información para ser almacenada en una BD.

Las principales ventajas del uso de una API REST son:

- **Separación entre el cliente y el servidor:** el protocolo REST separa totalmente la interfaz de usuario del servidor y el almacenamiento de datos.
- **Visibilidad, fiabilidad y escalabilidad:** la separación entre cliente y servidor tiene una ventaja evidente y es que cualquier equipo de desarrollo puede escalar el producto sin excesivos problemas.
- **La API REST siempre es independiente del tipo de plataformas o lenguajes:** la API REST siempre se adapta al tipo de sintaxis o plataformas con las que se estén trabajando.

3.- Llamadas a la API REST

Las llamadas a una API REST generalmente se realizan para realizar un **CRUD**. En informática, **CRUD** es el acrónimo de "Crear, Leer, Actualizar y Borrar" (del original en inglés: Create, Read, Update and Delete), que se usa para referirse a las funciones básicas en bases de datos o la capa de persistencia en un software.

Cada llamada o petición REST se implementan como peticiones HTTP, en las que:

- La URL representa el recurso `http://www.server.com/api/cursos`
- El método (HTTP Verbs) representa la operación
- El código de estado HTTP representa el resultado de la petición.

Las operaciones a realizar son las siguientes

- **Crear (POST):** Se deberá enviar en la petición el recurso que queremos añadir
- **Leer (GET):** Podemos realizar lecturas de todos los registros de una entidad o únicamente un registro de la entidad. Si no indicamos el id en la URL nos devolverá todos los registros de dicha entidad, si especificamos el id nos devolverá únicamente el registro con ese id.
- **Actualizar (PUT y PATCH):** Para realizar una actualización hay que **especificar en la URL la entidad y el id del elemento a modificar**. Además, **deberemos incluir en la petición el objeto con los valores que queremos actualizar**.

Hay que tener cuidado con la estructura de objeto que mandamos (los campos del objeto), pues tenemos dos métodos para la actualización y los resultados son diferentes por la forma en que actúan (uso de la idempotencia)

Los Métodos pueden ser:

- **PUT** según la ortodoxia REST, actualizar significaría cambiar **TODOS** los datos

- **PATCH** es un nuevo método estándar HTTP pensado para cambiar solo ciertos datos. Algunos frameworks de programación REST no lo soportan
- **Eliminar (DELETE)**: Para realizar una actualización hay que especificar en la URL la entidad y el id del elemento a eliminar.

4.- Clientes REST

Para interactuar con una API REST, se utilizan clientes REST, los cuales son programas o bibliotecas que permiten enviar y recibir peticiones HTTP a un servidor RESTful.

Un cliente REST es una herramienta que nos permite comunicarnos con una API RESTful. Funciona como intermediario, traduciendo nuestras solicitudes en peticiones HTTP que el servidor REST puede entender, y procesando las respuestas del servidor.

Los clientes REST se caracterizan por:

- **Seguir los principios REST**: Basan su funcionamiento en los principios RESTful, como la utilización de recursos, métodos HTTP y representaciones de estado.
- **Facilitar la comunicación**: Simplifican el proceso de interacción con APIs REST, ocultando la complejidad del protocolo HTTP y proporcionando una interfaz más amigable para el desarrollador.
- **Ser versátiles**: Pueden ser utilizados en diversos lenguajes de programación y plataformas, lo que los hace herramientas multipropósito.

El funcionamiento básico de un cliente REST se resume en los siguientes pasos:

1. **Solicitud del cliente**: El cliente REST envía una petición HTTP al servidor RESTful. Esta petición especifica el recurso que se desea consultar o modificar, el método HTTP a utilizar (GET, POST, PUT, DELETE) y los datos adicionales (si los hay).
2. **Procesamiento en el servidor**: El servidor RESTful recibe la petición, la interpreta y la procesa. En función del recurso y el método HTTP, el

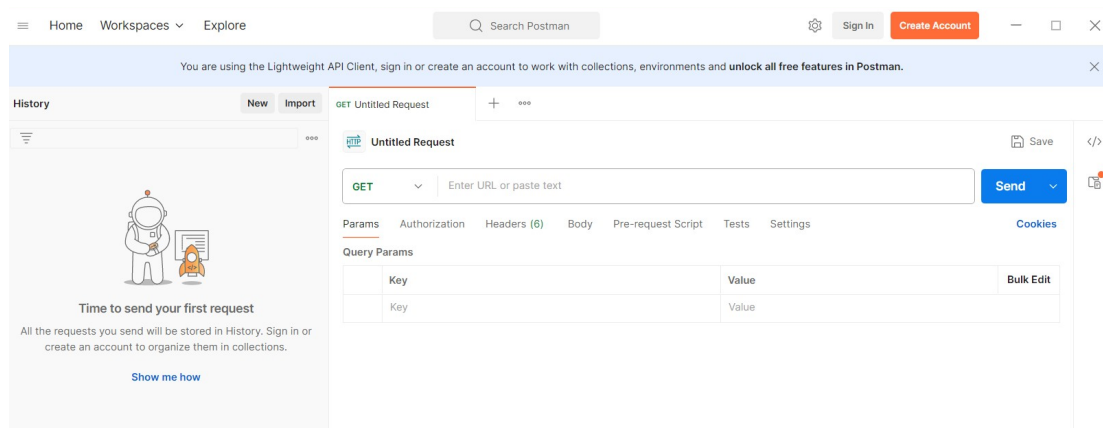
servidor realiza la acción correspondiente (obtener datos, crear un recurso, actualizar información, etc.).

3. **Respuesta del servidor:** El servidor RESTful envía una respuesta HTTP al cliente. Esta respuesta contiene el código de estado de la operación, los datos solicitados (si los hay) y otros metadatos relevantes.
4. **Interpretación por el cliente:** El cliente REST recibe la respuesta del servidor y la interpreta. En función del código de estado, el cliente sabe si la operación fue exitosa o si hubo algún error. Si se recibieron datos, el cliente los procesa y los utiliza en su aplicación.

Existen varias alternativas de clientes REST para poder probar API REST, yo voy a utilizar Postman.

- Descargamos Postman <https://www.postman.com/downloads/>
- Instalamos
- Seleccionamos la opción de **lightweight API cliente**

La interface es muy intuitiva

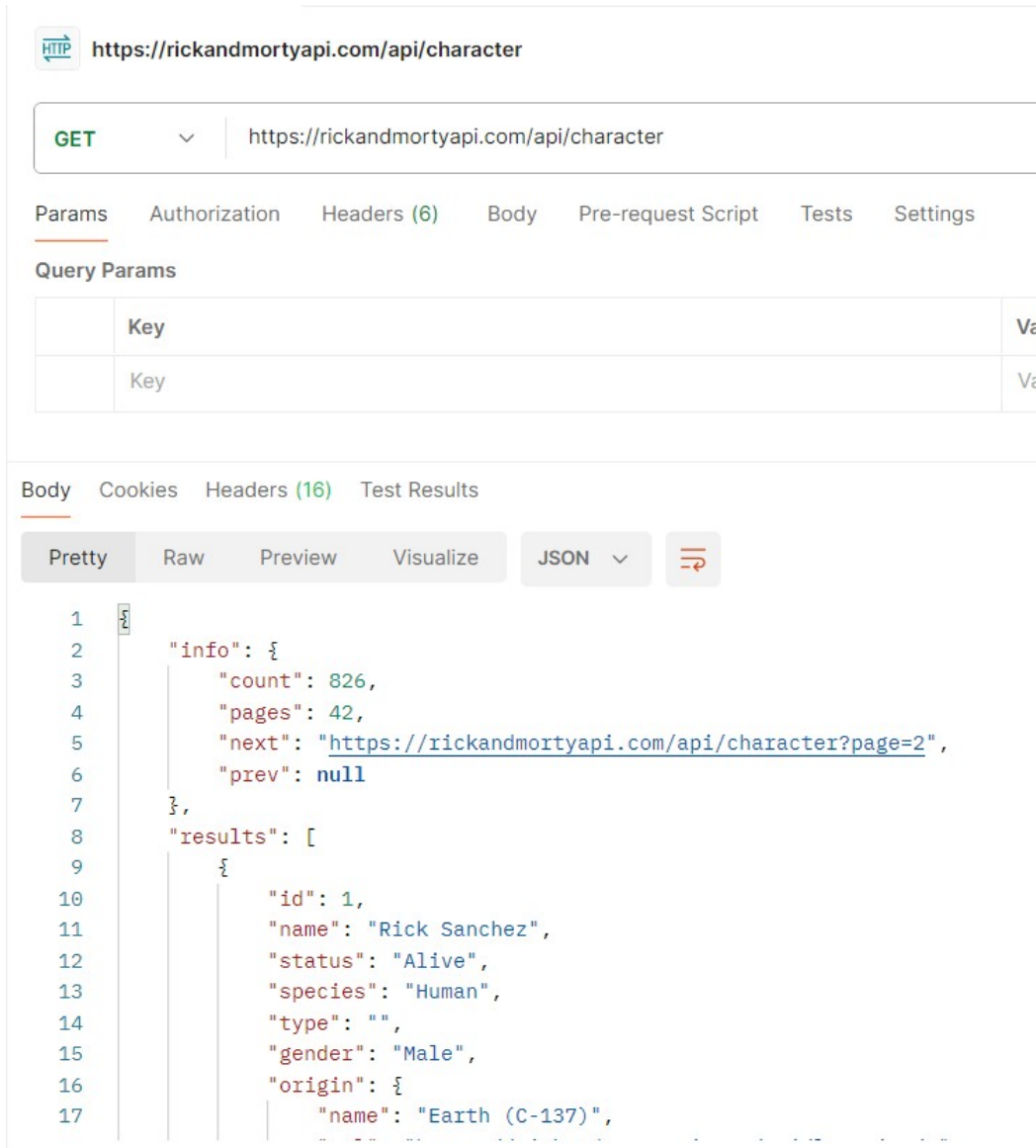


Para poder probarla necesitaríamos disponer de un API REST. Existen numerosas APIs públicas, generalmente suelen tener el inconveniente que solo nos dejan utilizar el método GET.

Vamos a probar la API de Rick and Morty <https://rickandmortyapi.com/>

Podemos consultar la documentación de la api para poder ver cómo funciona, nosotros **vamos a trabajar con el endpoint /character** que nos ofrece la información de los personajes de la serie.

Para una petición **get** es muy sencillo, seleccionamos del desplegable el método **get** e introducimos la url. El resultado será un objeto json con toda la información que nos devuelve el servidor.



The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** https://rickandmortyapi.com/api/character
- Query Params:** (Empty table)
- Body:** (Empty)
- Headers:** (16 headers listed)
- Test Results:** (Empty)
- Response Format:** JSON
- Response Body:**

```

1  {
2    "info": {
3      "count": 826,
4      "pages": 42,
5      "next": "https://rickandmortyapi.com/api/character?page=2",
6      "prev": null
7    },
8    "results": [
9      {
10       "id": 1,
11       "name": "Rick Sanchez",
12       "status": "Alive",
13       "species": "Human",
14       "type": "",
15       "gender": "Male",
16       "origin": {
17         "name": "Earth (C-137)",

```

También nos devolverá el código HTTP de la petición, en este caso 200

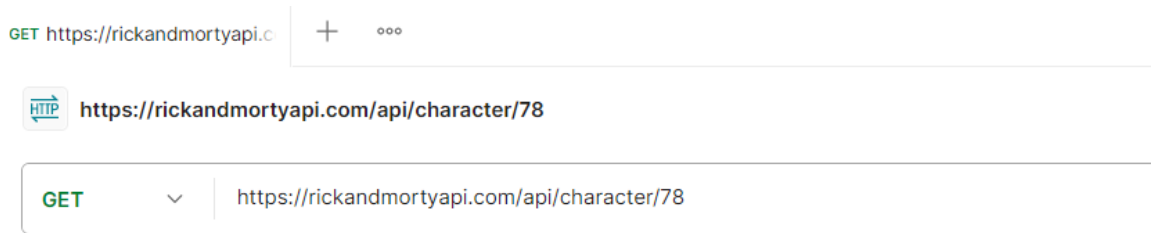
Key	Value	
Access-Control-Allow-Origin	*	
Age	4319	
Cache-Control	max-age	
Cache-Status	"Netlify"	
Content-Length	177	

Status: 200 OK Time: 47 ms Size: 944 B Save Response

200 OK

Standard response for successful HTTP requests. The actual response will depend on the request method used. In a GET request, the response will contain an entity corresponding to the requested resource. In a POST request the response will contain an entity describing or containing the result of the action.

Si quisiéramos solo un personaje utilizaríamos el método **get** indicando en la url el id del personaje.

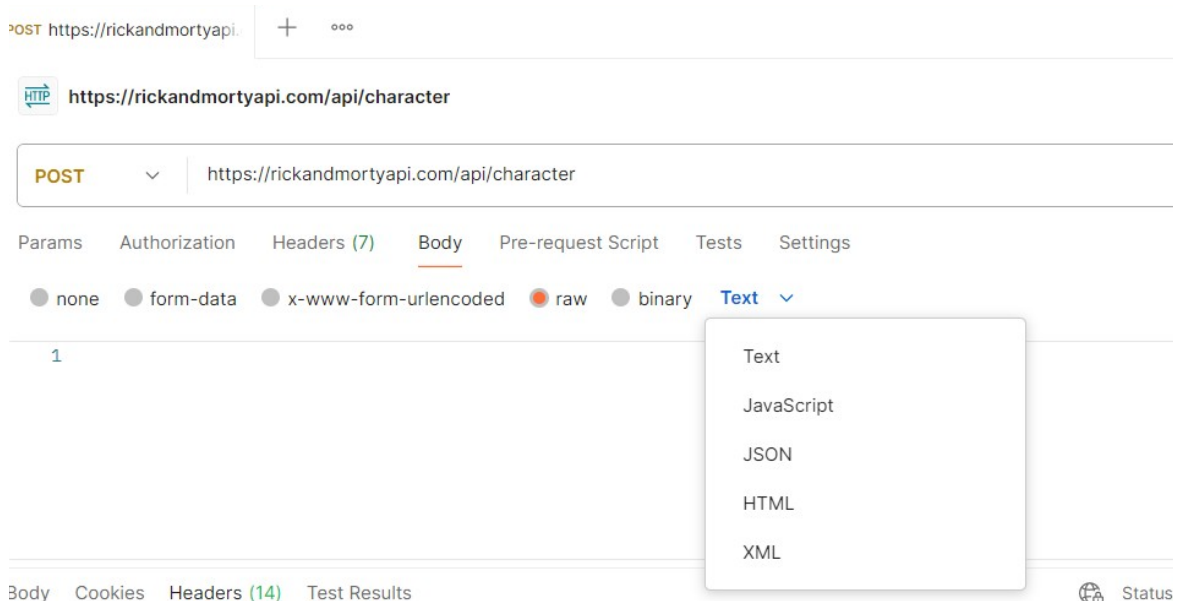


Si se introduce un endpoint que no exista se devolverá un error HTTP

La petición delete se realiza de la misma forma que un get para un elemento, es decir debemos seleccionar el método delete y añadir en el endpoint el id del elemento a borrar.

Las peticiones post, put y patch son diferentes ya que estas peticiones deben de mandar información al servidor. Para poder indicar el objeto JSON que mandaremos al servidor deberemos:

- Seleccionar el método post, put o patch
- Añadir la url
- Seleccionar la opción de Body (está debajo de la url)
- Seleccionar la opción de raw
- Seleccionar de la lista el formato JSON



Una vez realizado esto deberemos escribir el objeto json que queremos mandar al servidor.

Esta API al ser publica no permite realizar peticiones que modifiquen la información del servidor, es decir solo nos permite poder trabajar con el método get.

La API REST se implementa en el servidor (**backend**), nosotros estamos actuando de cliente (**front-end**). No vamos a diseñar una API REST en el servidor porque excede nuestros contenidos, no sería muy complicado ya que hoy en día casi todos los lenguajes de servidor llevan incorporadas funciones para poder realizar esta tarea. Para poder trabajar en clase vamos a utilizar una herramienta llamada **JSON Sever** que nos permitirá implementar una API REST en nuestro equipo y que utilizará como BD un fichero en formato JSON en el que se almacenará la información de las entidades y registros.

5.- JSON Server

Esta herramienta permite construir APIS REST de una manera muy sencilla y rápida, de hecho, es utilizada durante el proceso de desarrollo para poder testear en modo local las peticiones REST y de esta manera evitar las esperas innecesarias debidas a peticiones de servidores externos. Incluso se utiliza para poder empezar a desarrollar desde el front-end peticiones a una API REST que aún no ha sido implementada, ya que sí que sabemos las peticiones que vamos a realizar al servidor y que métodos y objetos debemos incluir en ellas.

Una vez finalizado el proceso de desarrollo y pasado a producción lo único que se debería de hacer es cambiar las URLs locales a las URLs del servidor externo real donde se haya diseñado la API REST.

JSON Server es un paquete npm que permite crear un servicio web REST JSON , respaldado por una base de datos simple. Está creado para desarrolladores **front-end** y ayuda a realizar todas las operaciones CRUD sin un prototipo o estructura de **backend** determinado.

Vamos a ver lo básico de la implementación de JSON Server así como de su funcionamiento para poder comprobar todo lo visto acerca de una API REST.

Toda la información acerca de json-Sever se encuentra en la url del proyecto <https://github.com/typicode/json-server/tree/v0>

Las acciones básicas para poder trabajar son las siguientes:

- **Para instalar** JSON Server deberemos instalar el paquete correspondiente con npm. Para ello deberemos tener instalado **Node.js** en nuestro equipo

```
npm install -g json-server
```

JSON Server trabaja con una **base de datos no relacional** muy sencilla, para ello lo que se utiliza es un fichero en formato JSON. Por defecto nos recomienda crea un archivo llamado **db.json** con algunos datos

```
{
  "posts": [
    { "id": 1, "title": "json-server", "author":
"typicode" }
  ],
  "comments": [
    { "id": 1, "body": "some comment", "postId": 1 }
  ],
  "profile": { "name": "typicode" }
}
```

Como podemos ver lo que se crea es un objeto JSON en el que cada propiedad es el nombre de una entidad (tabla del modelo relacional) y su value es un array con los datos de esa entidad (registros del modelo relacional), cada elemento tendrá la estructura que permita almacenar los datos deseados en el par name:value de JSON (campos del modelo relacional).

- **Iniciar JSON Server:** Para iniciar el servidor deberemos ejecutar en la consola el comando json-server - -watch con el nombre de la BD que queramos utilizar

```
json-server --watch db.json
```

En ese momento el servidor estará disponible en la url **http: // localhost: 3000**.

Para poder probarla lo único que deberemos hacer es realizar las peticiones REST adecuadas.

Si realizamos la siguiente petición <http://localhost:3000/posts/1> obtendremos el siguiente resultado

```
{ " id " : 1 ,  
  " title " : " json-server " ,  
  " author " : " typicode " }
```

JSON Server implementa las siguientes características

- Todas las entidades deben tener definido un id en el fichero json
 - Si realizamos POST, PUT, PATCH o DELETE, los cambios serán automáticos y se guardaran en db.json.
 - El body de las peticiones debe ser un objeto JSON, al igual que la devolución de una petición GET.
 - Los valores de identificación no son mutables. Cualquier valor id en el cuerpo de la solicitud PUT o PATCH será ignorado. Solo se respetará un valor establecido en una solicitud POST.
 - Una solicitud POST, PUT o PATCH debe incluir un encabezado **Content-Type: application/json** para usar el JSON en el cuerpo de la solicitud. De lo contrario, devolverá un código de estado 2XX, pero sin que se realicen cambios en los datos.
- **Rutas:** Según el archivo db.json que se crea por defecto, aquí están todas las rutas predeterminadas.

```
GET /posts  
GET /posts/1  
POST /posts  
PUT /posts/1  
PATCH /posts/1  
DELETE /posts/1
```
 - **Filtros:** Los filtros están pensados para hacer peticiones GET a la API REST en las que la consulta no está basada en la id de la entidad, sino que está basada en uno varios campos de la entidad

Para acceder a propiedades profundas

```
GET /posts?title=json-server&author=typicode
```

```
GET /posts?id=1&id=2
```

```
GET /comments?author.name=typicode
```

6.- Json-Server y Postman

Con **json-server** ya podemos utilizar una API REST completa, es decir podemos ejecutar todos los métodos, y con **Postman** podemos comprobar si realmente se están realizando las peticiones como esperamos.

Ahora podemos modificar los datos de nuestra bd, cosa que los apis públicas es prácticamente imposible.