ÍNDICE

1. JSON 2
Reglas de Sintaxis con JSON 2
Envío de datos JSON 3
2. Cookies. 3
3. WebStorage. 4
4. SetInteval y SetTimeout. 5
5. Promesas en JavaScript. 6
Los métodos .then() .catch() .finally() 7
6. Asincronía. 9
7. API Rest en JavaScript 10
Arquitectura de una API REST 11
Peticiones REST: Request 12
Respuestas REST: Response 12
Implementación de una API REST 13
Estructura de un endpoint 14
Convenciones establecidas 14
8. Fetch API en JavaScript. 15
A. ¿Cómo hacer un GET utilizando la API Fetch? (Traer Datos) 16
B. ¿Cómo hacer un POST utilizando la API Fetch? (Enviar Datos) 16
C. ¿Cómo hacer un PUT utilizando la API Fetch? (Actualizar Datos) 16
D. ¿Cómo hacer un DELETE utilizando la API Fetch? (Borrar Datos) 17

1. JSON

En los objetos js las claves no van entre comillas, aunque sean un string, mientras que en json SI

JSON son las siglas de "**JavaScript Object Notation**". Un archivo JSON, tiene como extensión los *.json*, además los datos que contiene son representados en un par **llave:valor**, igualmente que un objeto JavaScript tradicional.

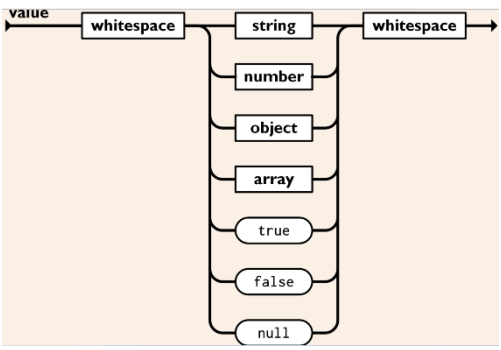
Sin embargo, JSON y los objetos de JavaScript no son exactamente lo mismo. La diferencia principal es que la llave en JSON son los valores, aparte de los tipos número y nulo deben ir entre comillas dobles. no

```
{
  "llave1": "valor1",
  "llave2": "valor2",
  "llave3": "valor3",
  "llave4": 7,
  "llave5": null,
  "favAmigos": ["Kolade", "Nithya", "Dammy", "Jack"],
  "favJugadores": {"uno": "Kante", "dos": "Hazard", "tres": "Didier"}
}
```

Los tipos de datos undefined, function y date NO SON COMPATIBLES con JSON.

Reglas de Sintaxis con JSON

- Todos los datos del archivo deben estar rodeados de llaves {} si quieres representar un objeto y entre corchetes cuadrados si es un Array [].
- Las comillas **simples** no están permitidas.
- La llave en cada JSON debe ser única y debe estar entre comillas dobles.
- Los números no deben ir entre comillas dobles, de lo contrario se tratarán como cadenas de texto.
- El tipo de dato null no debe ir entre comillas dobles.
- Los valores booleanos solo pueden ser verdaderos o falsos.
- Cada par llave:valor debe terminar con una coma, excepto el último elemento
- Un solo objeto dentro de un arreglo debe terminar también con una coma.



Envío de datos JSON

Los datos JSON no se pueden transmitir a través del navegador en forma clave:valor . Para manipular los datos JSON cada lenguaje de programación tiene métodos específicos.

- JSON.parse() convierte los datos JSON en objetos. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse

- JSON.stringify() convierte el par clave:valor de un objeto en datos JSON. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify

Este último método devuelve una cadena de texto JSON. Posteriormente ya se modría manipular a través del DOM

Enlaces de interés:

https://json.org/json-es.html

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/JSON

Nota: Para probar los datos podríamos usar una web como:

https://jsonplaceholder.typicode.com/

Ejemplo:

```
<h2>Aquí están los datos del JSON:</h2>
<div id="json"></div>

const JSONData =
    '{"name": "Kolade", "favFriends": ["Kolade", "Nithya", "Rocco", "Jack"], "from": "Africa"}';
  try {
    const JSONString = JSON.parse(JSONData);
    const JSONDisplay = document.querySelector("#json");
    JSONDisplay.innerHTML = JSONString.name + ", [" + JSONString.favFriends + "], " + JSONString.from;
  } catch (error) {
    console.log("No puedo parsear JSON Data");
  }
```

2. Cookies.

Mecanismo para almacenar información cada vez que se accede a una página. Cada vez se usa menos, sustituyéndola por los mecanismos que aporta webStorage.

https://developer.mozilla.org/es/docs/Web/API/Document/cookie

https://developer.mozilla.org/es/docs/Web/API/Document/cookie#ejemplos

3. WebStorage.

Web Storage es una **API** de almacenamiento web que proporciona los mecanismos mediante los cuales el navegador puede almacenar información de tipo clave-valor de una forma mucho más intuitiva que utilizando cookies.

Cuando construimos una app, normalmente trabajamos con algún tipo de información, por ejemplo, si le pedimos a un usuario que introduzca una ciudad para ver qué tiempo hace allí. Pero si no almacenamos esa información en algún sitio, ésta se perderá cuando recarguemos la página. La solución sería o almacenarlo en una base de datos o en WebStorage

Cuando hablamos de **Web Storage** disponemos, a su vez, de dos mecanismos de almacenamiento web:

- **sessionStorage**: que permite almacenar información mientras el navegador está abierto; es decir, mientras dura la sesión de la página. Todas las páginas que tienen el mismo origen (dominio y protocolo) pueden acceder a estos datos indistintamente.

```
//window.sessionStorage: almacena los datos durante una sesión
//si se cierra la ventana o el navegador, desaparecen
```

- **localStorage**: similar a *sessionStorage* pero los datos se mantienen aún a pesar de que cerremos el navegador.

```
//window.localStorage: almacena datos SIN fecha de expiración
```

En primer lugar, lo más recomendable es comprobar si el navegador que vamos a utilizar soporta **Web Storage**. Para hacer esto usamos:

```
//Comprobar si el navegador soporta webstorage
if (typeof(Storage) !== "undefined"){
    alert ("El navegador soporta webStorage");
}else{
    alert("El navegador NO soporta WebStorage");
}
```

A continuación, utilizaremos los diferentes métodos que nos permitirán trabajar con este tipo de almacenamiento:

- **setItem**: permite crear un elemento de almacenamiento web.
- **getItem**: permite consultar un elemento.
- **removeItem y clear**: permite eliminar información de un elemento o el elemento completo.

La propiedad ***length*** representa el número de cosas (items) que tenemos almacenados en localStorage actualmente.

Hemos dicho entonces que en localStorage almacenamos datos. Cada dato será un *item* que tendrá una **key** y un **value**, muy parecido a la estructura de un objetos en JavaScript (JS). Con la particularidad de que un item debe ser siempre un string. Eso no significa que no podamos crear arrays, objetos, o cualquier otro data type. Simplemente significa que cualquier data type debe ir entre comillas como un string para poder ser almacenado en local storage.

Ejemplos:

```
localStorage.setItem('fullName', 'Mi nombre');

let name = localStorage.getItem('fullName');

localStorage.removeItem('fullName');

localStorage.clear()
```

Otro Ejemplo: Almacenar JSON en LocalStorage. Dado el siguiente Array de Objetos:

```
const toDos = [

    { task: "play dungeons & dragons", author: "Will" },

    { task: "escape from Demogorgon", author: "Will" },

    { task: "go to the arcade", author: "Will" }

];
```

En primer lugar lo convertimos a JSON String con stringify() para seguidamente ya almacenarlo en localStorage:

```
localStorage.setItem('toDosList', JSON.stringify(toDos));
```

lo guardas con JSON.stringify() para mandarlo como string, al ser lo que soporta webstorage,
y se recoge con JSON.parse() para cambiarlo de string a objeto json operable

Enlaces:

https://developer.mozilla.org/es/docs/Web/API/Window/localStorage

https://developer.mozilla.org/es/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API

https://mdn.github.io/dom-examples/web-storage/

https://html.spec.whatwg.org/multipage/webstorage.html

4. SetInteval y SetTimeout.

Se utilizan para simular la asincronía en las aplicaciones de JavaScript.

setInterval(función,tiempo) → Ejecuta dicha función indefinidamente cada n milisegundos.

setTimeout(función,tiempo) → Ejecuta una función dentro de n segundos.

Por ejemplo, se podría programar un reloj en el navegador.

```
Let temporizador = setInterval(() => {
        console.log(new Date().toLocaleTimeString());
},1000);
```

Hay otras funciones que cancelan las anteriores.

clearTimeout(temporizador) // importante para cancelar un temporizador ha de estar almacenado obligatoriamente en una variable.

clearInterval(variable_intervalo) // cancelaría un intervalo.

5. Promesas en JavaScript.

Una promesa es un objeto que representa un valor futuro, pero no tenemos certeza de cuánto va a tardar y cuándo va a terminar la operación.

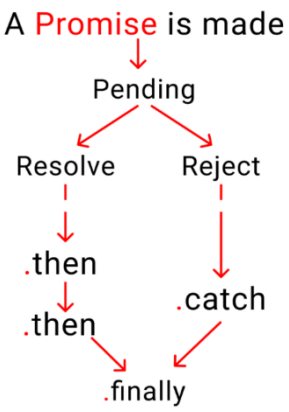
La ventaja es que nos va a proporcionar una API, para que cuando la operación haya terminado, poder obtener el valor o manejar la excepción en caso de que se produzca un error.

Invocamos las promesas con new Promise con dos parámetros:

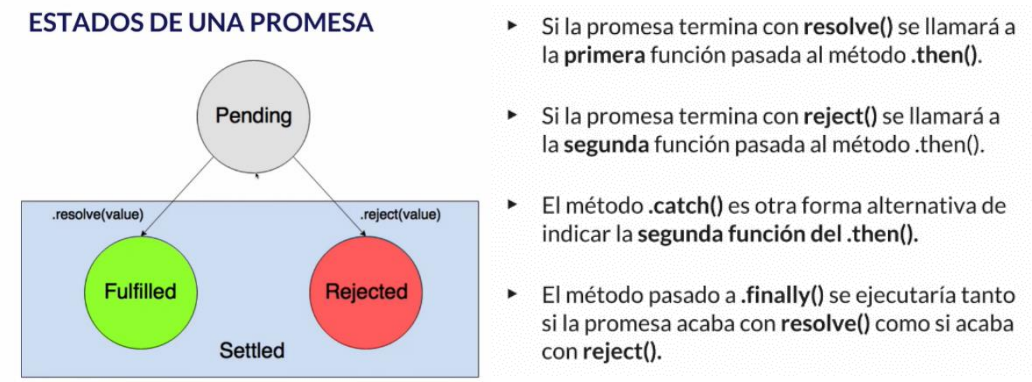
- Callback **resolve** --> llamaremos cuando la operación se procese correctamente.
- Callback **reject** --> usaremos cuando se produzca un error o cuando nosotros consideremos que se ha producido un error llamaré a resolve o reject cuando corresponda según el flujo.

Resolve y Reject serían métodos estáticos.

El esquema sería el siguiente:



Por tanto, una promesa tiene tres estados: **Pendiente, resuelto y rechazado**.



Los métodos .then() .catch() .finally()

Métodos	Descripción
.then(resolve)	Ejecuta la función callback resolve cuando la promesa se cumple.
.catch(reject)	Ejecuta la función callback reject cuando la promesa se rechaza.
.then(resolve,reject)	Método equivalente a las dos anteriores en el mismo .then().
.finally(end)	Ejecuta la función callback end tanto si se cumple como si se rechaza.

Ejemplo:

Partiendo de la siguiente información

```
const peliculas = [
  {
    id: 1,
    title: "titulo 1",
    sinopsis: "contenido titulo1",
  },
  {
    id: 2,
    title: "titulo 2",
    sinopsis: "contenido titulo 2",
  },
];
```

Necesitamos realizar una búsqueda de la información de ese Array de Objetos. Para ello me creo una promesa (new Promise) a la que hay que pasar como parámetros resolve y reject (o cualquier nombre, teniendo presente lo que simbolizan).

Resolve simboliza la consecución de la promesa.

Reject simboliza la no consecución de la promesa.

Cuando vamos a consumir una promesa debemos de tener presentes los métodos a los que se puede invocar para traernos el resultado de la acción a la que se le asignó una promesa.

.then()→ Trabajaríamos con los datos obtenidos cuando la promesa ha sido satisfecha correctamente.

.catch() → Trabajaríamos con los datos obtenidos cuando la promesa no ha sido satisfecha.

.finally() → Aquí se accedería siempre tanto si la promesa ha sido satisfecha como si no lo ha sido.

return

```
const buscarPeliculasId = (id) =>
  new Promise((resolve, reject) => {
    const pelicula = peliculas.find((item) => item.id === id);
    if (pelicula) {
      resolve(pelicula);
    } else {
      reject("No se encontró el id " + id);
    }
  });
// para consumir la promesa
buscarPeliculasId(2)
  .then((pelicula) => console.log(pelicula))
  .catch((error) => console.error(error));
    }
  });
};
```

Me gustaría recalcar que una de las ventajas que tienen las promesas es el poder trabajar con una información sin necesidad de tenerla de antemano.

Las promesas no llegan a tener código como el callback hell pero aun así tienen sus limitaciones y complicaciones:

Si en nuestro ejemplo asíncrono (he colocado un setTimeout para ejemplificar el retardo en una petición) cuando consumo la promesa realizo llamadas unas dentro de otras tendría un código como el siguiente, que sin llegar a ser CallBack, pero dificulta su comprensión.

```
buscarPeliculasId(1)
  .then((pelicula) => {
    console.log(pelicula);
    return buscarPeliculasId(2);
  })
  .then((pelicula) => {
```

```
  console.log(pelicula);
  return buscarPeliculasId(4);
})
.then((pelicula) => {
  console.log(pelicula);
  return buscarPeliculasId(5);
})
.catch((error) => console.error(error));
```

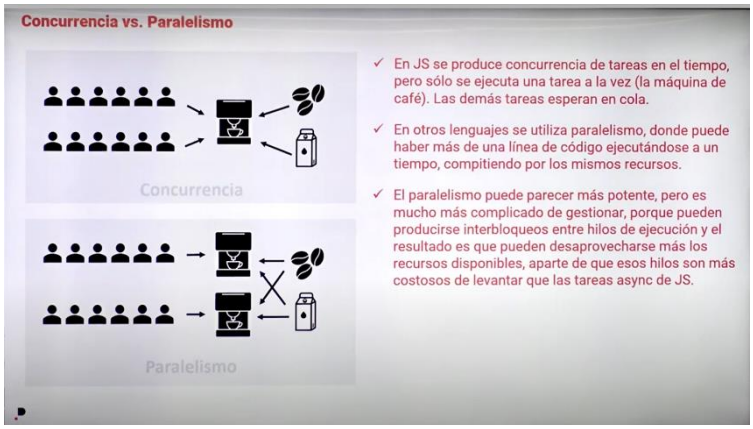
Para mejorar lo anterior necesitaríamos de async/await (***se verá posteriormente en clases posteriores***) pero como dato diría que:

- async : La declaración de función async define una función asíncrona, la cual devuelve una AsyncFunction.
- await : El operador await es usado para esperar a una Promise. Sólo puede ser usado dentro de una función async function.

6. Asincronía.

La asincronía es uno de los pilares fundamentales de JavaScript. Es un concepto donde intervienen. Teniendo presente que en JavaScript sólo se ejecuta una tarea a la vez, tendríamos:

- Concurrencia: cuando dos o más tareas progresan simultáneamente.
- Paralelismo: cuando dos o más tareas se ejecutan, literalmente, a la vez, en el mismo instante de tiempo.



Una buena entrada, donde profundizar en esto sería la siguiente:

https://lemoncode.net/lemoncode-blog/2018/1/29/javascript-asincrono

y un vídeo-conferencia también explicativo de la programación Asíncrona en JavaScript sería:

Curso Programación asíncrona en JavaScript. - YouTube

7. API Rest en JavaScript

Una API no es más que, por una parte, la definición de las funciones que puede realizar un software y por otra parte la metodología que podemos utilizar para comunicarnos con él (junto con la información que le debemos transmitir).

El término **API** deriva de ***Application Program Interface*** y se define como la implementación de una metodología de comunicación entre varios componentes o sistemas. Una API permite que un sistema, una aplicación o un componente intercambie información con otros.

Existen dos alternativas populares para consumir o comunicarnos una API REST desde javascript; **Fetch API** y la Biblioteca Axios.

La mayor parte de las APIs web implementan la arquitectura ***CRUD***, que permite crear registros, obtenerlos, actualizarlos y eliminarlos. Una API que implementa las operaciones CRUD también suele recibir el nombre de ***API REST*** si soporta los siguientes métodos HTTP, correspondiéndose con sus respectivos verbos CRUD:

Acción CRUD	Método HTTP	Descripción
Creación	POST	Creación de un registro
Lectura	GET	Obtención de un registro
Actualización	PUT o PATCH	Actualización de un registro
Borrado	DELETE	Borrado de un registro

Las APIs REST implementan usa serie de estándares que dan forma a la arquitectura REST, que es una de las más utilizadas a día de hoy cuando desarrollas aplicaciones web o aplicaciones móviles.

REST es el acrónimo de **Representational State Transfer**, que describe los estándares de una arquitectura de servicios web. En concreto, se define el modo en el que se deben compartir los datos entre diferentes sistemas que implementan la arquitectura REST, también conocida como **RESTful**.

La arquitectura ***REST*** no es ni un lenguaje ni un framework, sino un concepto que define una metodología a seguir durante el desarrollo de los diferentes sistemas de una aplicación.

Una *API web* consta en general de dos acciones básicas, que son la ***petición*** al servidor y la ***respuesta*** del mismo. Cuando se trata de la comunicación entre dos sistemas, la parte visual no es relevante, por lo que, cuando se realiza una petición a un endpoint del servidor, este suele responder con datos en un formato determinado, como por ejemplo *XML* o *JSON*.

Arquitectura de una API REST

Una API REST debe cumplir una serie de principios o restricciones para que pueda ser considerada como una API RESTful:

- ***Arquitectura cliente servidor***: El cliente será el encargado de realizar una petición al servidor, que consiste en una acción a realizar junto con los datos necesarios para procesar la petición. El servidor será el encargado de procesar la petición del cliente y proporcionar una respuesta.
- ***Identificación de recursos***: La identificación de cada elemento debe ser uniforme para todos los recursos disponibles en una API. En el caso de una API HTTP, se usa la URL para identificar a los recursos. Por ejemplo, en el caso de un servicio de streaming de cine, identificaremos a las diferentes películas mediante la URL.
- ***Interacciones stateless***. Es decir, que el servidor no debería necesitar hacer referencia a peticiones anteriores para procesar la petición actual, no siendo necesario el uso de sesiones. Toda la información de una petición deberá estar descrita en la propia petición. De ahí deriva el termino stateless, que significa «sin estado», de modo que no es necesario mantener un estado en el servidor para procesar próximas peticiones.
- ***Sistema de capas***: Se acepta el uso de diferentes capas que pueden actuar durante las peticiones. Por ejemplo, el cliente puede no siempre conectarse al mismo servidor, ya que las peticiones pueden ser redireccionadas a diferentes servidores mediante balanceadores de carga. Los balanceadores de carga o load balancers suelen redirigir las peticiones a varios servidores en función de lo cargados que estén, acelerando así el tiempo de respuesta.
- ***Uso de caché***: El servidor puede mantener la respuesta a las peticiones en caché, por lo que las acciones que se llevarán a cabo pueden no ser siempre las mismas.

Vamos a centrarnos ahora en las APIs REST en el contexto del desarrollo de aplicaciones web. Como sabes, las URLs suelen comenzar por HTTP o por HTTPS, dependiendo del protocolo a utilizar.

El protocolo HTTP utiliza conexiones TCP para conectarse a un puerto del servidor, que suele ser el puerto 80 si usamos el protocolo HTTP o el puerto 443 si usamos el protocolo HTTPS. Una petición constará siempre de una **URL**, un método, una serie de **cabeceras** y un cuerpo o **body**. Por ejemplo, cuando realizas una petición mediante tu navegador estás realizando peticiones GET de obtención de datos o recursos. Sin embargo, cuando envías un formulario, realizarás una petición POST de envío de datos desde tu navegador, que permitirá la creación de recursos o datos en el servidor.

Al recibirla, el servidor procesará la petición recibida y enviará una respuesta que constará de un estado y de un body como elementos fundamentales. Por ejemplo, cuando accedes a una página de error con tu navegador, seguramente estés acostumbrado a ver el estado 400. Cuando una petición se procesa correctamente, el estado recibido es el 200 aunque no se muestre explícitamente en tu navegador.

A continuación, vamos a explicar en detalle tanto las peticiones como las respuestas.

Peticiones REST: Request

Las peticiones HTTP pueden ser de diferentes tipos, definidos mediante su acción. A continuación puedes ver una lista con los tipos de peticiones HTTP más comunes que se suelen implementar en una API:

- ***GET**: El método GET se utiliza para obtener recursos mediante una petición a la API, implicando la lectura de uno o más recursos en el servidor en base a los parámetros especificados en la URL. Una petición GET nunca alterará el estado de los datos en el servidor, siendo únicamente una operación de lectura.*
- ***POST**: El método POST se utiliza para crear recursos mediante una petición a la API, implicando la creación de un recurso en el servidor en función de los datos enviados al mismo. Cuando te registras en una red social o cuando creas contenido en ellas, realizas una petición POST. Este tipo de petición suele implicar una escritura de datos en un archivo del servidor o en una base de datos.*
- ***PUT**: El método PUT se utiliza para actualizar recursos mediante una petición a la API. Este tipo de petición suele implicar tanto la lectura como la escritura de un recurso en el servidor. Una misma petición PUT realizada múltiples veces de forma consecutiva producirá siempre el mismo resultado, motivo por el que se dice que las peticiones PUT son idempotentes.*
- ***PATCH**: El método PATCH también se utiliza para actualizar recursos mediante una petición a la API, aunque a diferencia del método PUT, el método PATCH se usa para actualizar ciertas propiedades de un recurso, en lugar de reemplazarlas todas.*
- ***DELETE**: El método GET se utiliza para obtener recursos mediante una petición a la API implicando la eliminación un recurso en el servidor, identificado mediante los parámetros proporcionados en la URL.*

Los métodos definidos en las peticiones HTTP de esta lista se corresponden con los del paradigma **CRUD**, que son los métodos de lectura, escritura, eliminación y actualización utilizados en una base datos.

Respuestas REST: Response

Cuando el servidor recibe una petición HTTP y la procesa, devolverá una respuesta HTTP que incluirá los siguientes elementos básicos:

- ***Cabeceras**: También conocidas como headers, son metadatos acerca del contenido devuelto y del resultado de la petición.*
- ***Body**: También conocido como cuerpo, el body incluye los datos devueltos por la petición. Si por ejemplo hemos pedido los datos de un usuario, estos se devolverán en el body.*
- ***Estado**: Las respuestas HTTP incluyen un código numérico de respuesta, también conocido como status code, que en función de su valor, proporcionan información acerca del resultado de la petición.*

Además de proporcionarnos información acerca del resultado de una operación, el código de estado también nos dice si se requieren más operaciones o no para completar la operación. Los

códigos de estado suelen constar de tres dígitos, que pueden clasificarse como los siguientes tipos:

- Información: Los códigos de respuesta que comienzan por 1 nos proporcionan información acerca de una petición, sin tratarse de la respuesta a la petición en sí misma.
- Éxito: Los códigos de respuesta que comienzan por 2 indican que la operación que se corresponde con la petición ha sido realizada con éxito. Por ejemplo, cuando accedes a una página con éxito, el código que se devuelve es el 200.
- Redirección: Los códigos de respuesta que comienzan por 3 indican que se ha producido una redirección de un recurso a otro. Por ejemplo, el código 301 «Moved Permanently» se usa para redireccionar un página a otra URL.
- Error en le petición: Los códigos de respuesta que comienzan por 4 se usan para indicar un error en la petición o durante el proceso de la misma. Por ejemplo, si una página o recurso al que se accede no existe en la base de datos del servidor, se devuelve el código 400 «Not Found».
- Error en el servidor: Los códigos de respuesta que comienzan por 5 se usan para indicar que se ha producido un error en el servidor. Por ejemplo, el código de error 500 «Internal Server Error» es devuelto automáticamente por muchos servidores cuando se produce un error inesperado durante la ejecución del código del propio servidor.

En cuanto a los códigos de respuesta que se corresponden con las operaciones REST de una API, deberían ser los siguientes, en función del tipo de petición:

Petición	Código	Descripción
GET	200	OK
POST	201	Created
PUT	200	OK
PATCH	200	OK
DELETE	200	OK
DELETE	202	Accepted
DELETE	204	No content

Los códigos de esta lista se usan cuando la petición se ha procesado correctamente. El código 200 indica que la petición se ha completado con éxito, siendo usado en peticiones GET, PUT, PATCH y DELETE. El código 201 se usa únicamente en peticiones POST, indicando que el recurso se ha creado correctamente.

Tal y como ves, las peticiones DELETE pueden devolver varios códigos de respuesta válidos. El código 200 se suele usar cuando el recurso se ha borrado correctamente, mientras que el 202 se usa cuando el recurso se ha marcado para ser borrado, pudiendo este haberse borrado o no. Finalmente, el código 204 se usa cuando el recurso indicado no existe.

Implementación de una API REST

Cuando creas una API en un servidor, crearás *endpoints*, que son las URLs a las que deben apuntar las peticiones HTTP. Dado que sería impracticable crear endpoints individuales para

cada uno de los recursos de una aplicación, la creación de rutas suele facilitarse con ciertas herramientas cuando usas algún framework, pudiendo definir su estructura.

Estructura de un endpoint

Un endpoint suele aceptar uno o más verbos GET, POST, PUT, PATCH o DELETE, estando compuestos de una raíz, una ruta o path y opcionalmente de una consulta o query string:

- Raíz: Lar raíz es la URL que se establece como base para una API, que suele constar del protocolo (HTTP / HTTPS), del dominio y de un número de versión. Por ejemplo, la URL https://dominio.com/v2 consta del protocolo HTTPS, del dominio dominio.com y del número de versión v2.
- Ruta: La ruta es la estructura o localización de un recurso. Por ejemplo, la ruta /peliculas identificaría a un conjunto de películas, mientras que la ruta /peliculas/2 identificaría a la película cuyo identificador es 2. Sin embargo, es habitual usar estructuras somo /peliculas/{id} a la hora de definir una ruta, dependiendo del framework que se utilice.
- Consulta: Son los pares de valores GET definidos en una URL, que suelen usarse para ordenar, filtrar o paginar los valores obtenidos. Por ejemplo, la consulta ?page=2 suele usarse para obtener la segunda página de un conjunto de recursos paginados.

El resultado de los ejemplos que hemos definido darían lugar a los endpoints https://dominio.com/v2/peliculas?page=2 o https://dominio.com/v2/peliculas/2. El primer endpoint devolvería todas las películas que estén en la segunda página del conjunto de películas. Es decir, que si las páginas contienen 10 elementos, se mostrarían los elementos que van del 10 al 20. Por otro lado, el segundo endpoint devolvería únicamente la película cuyo identificador es 2.

Convenciones establecidas

Existen una serie de convenciones o estándares que deberías aplicar cuando crees una API REST. El uso de estándares hace que el código sea más sencillo de mantener y de testear.

Por ejemplo, en cuanto a la sintaxis de las rutas, estas únicamente deberían estar compuestas de caracteres en minúscula y de guiones, siguiendo la notación kebab-case. El uso de caracteres en mayúscula o de guiones bajos está mal considerado.

Los nombres de las rutas deberían estar siempre en plural, aunque identifiquen a un único elemento:

- La ruta /mensajes es la que se usará para obtener varios mensajes.
- Lara ruta /mensajes/2 es la que se usará para obtener el mensaje cuyo identificador es 2. No deberías usar la ruta /mensaje/2, ya que podría confundir tanto a desarrolladores como a terceros que creen aplicaciones que consuman la API.

Las rutas de un endpoint deberían contener únicamente nombres el lugar de verbos:

- La ruta que se usa para obtener un mensaje, debería ser /mensajes/5 en lugar de /mensajes/2/get, ya que la acción ya se especifica mediante el tipo de petición, que debería ser GET.
- La ruta utilizada para crear un mensaje, debería ser /mensajes en lugar de /mensajes/crear, ya que la acción ya se especifica mediante con la petición POST.
- La ruta usada para borrar un mensaje, debería ser /mensajes/2 en lugar de /mensajes/2/eliminar, puesto que la acción ya se debería especificar en el tipo de petición DELETE.
- La ruta que se usa para actualizar un mensaje, debería ser /mensajes/2 en lugar de /mensajes/2/actualizar, dado que la acción ya se especifica mediante el tipo de petición PUT o PATCH.

Una ruta nunca debería contener la extensión del archivo devuelto, con independencia de que devuelva un archivo JSON o XML. Por ejemplo, una ruta válida sería /mensajes, mientras que el uso de /mensajes.json está mal considerado.

Estas reglas son solamente recomendaciones y no normas estrictas, pero su uso son una buena práctica fácilmente reconocible por parte de los desarrolladores con más experiencia. El uso de estándares hace que una API sea más consistente, especialmente cuando varios desarrolladores trabajan en ella.

El texto anterior ha sido sacado de la guía:

Qué es una API REST | Neoguias

8. Fetch API en JavaScript.

La API ***Fetch*** proporciona una interfaz para obtener recursos (incluso a través de la red). Parecerá familiar a quien sea que haya usado ***XMLHttpRequest***, pero proporciona un conjunto de características más potentes y flexibles.

En pocas palabras la Fetch API es una alternativa moderna que nos permite interactuar con APIs y obtener los datos a nuestra aplicación, de una forma parecida a lo hacíamos con AJAX pero de una manera mucho más sencilla.

El único parámetro requerido de fetch() es una url. El método por defecto en este caso es GET.

```
Fetch('url')
```

```
Fetch('url ', {
        method: ('GET/POST/PUT/DELETE ', //optional
        headers: { //optional
        Content-Type ': application/json ' //optional
                ...
        },
        body: formData //optional
})
```

A. ¿Cómo hacer un GET utilizando la API Fetch? (Traer Datos)

El método GET se usa para recuperar datos del servidor. Este es un método de solo lectura, por lo que no tiene riesgo de mutar o corromper los datos.

```
fetch('https://jsonplaceholder.typicode.com/users')
    .then(response => response.json())
    .then(json => console.log(json))
```

B. ¿Cómo hacer un POST utilizando la API Fetch? (Enviar Datos)

El método POST envía datos al servidor y crea un nuevo recurso.

```
fetch('https://jsonplaceholder.typicode.com/todos', {
        method: 'POST',
        body: JSON.stringify({
           name: "Taylor",
           surname: "Swift"
        }),
        headers: {
           "Content-type": "application/json"
        })
    .then(response => response.json())
    .then(json => console.log(json))
```

C. ¿Cómo hacer un PUT utilizando la API Fetch? (Actualizar Datos)

El método PUT se usa con mayor frecuencia para actualizar un recurso existente.

```
fetch('https://jsonplaceholder.typicode.com/todos', {
        method: 'PUT',
        body: JSON.stringify({
           id: 1,
           name: "Taylor",
           surname: "Swift"
        }),
        headers: {
           "Content-type": "application/json"
        })
```

```
    .then(response => response.json())
    .then(json => console.log(json))
```

D. ¿Cómo hacer un DELETE utilizando la API Fetch? (Borrar Datos)

El método DELETE se usa para eliminar un recurso.

```
fetch('https://jsonplaceholder.typicode.com/users/id', {
   method: 'DELETE'
});
¿Cómo capturo los datos de un fetch? ¿Promises? ¿Asyn Await?
```

Importante:

El método ***Fetch devuelve una promesa,*** lo que quiere decir que podemos utilizar el método .then cuando la promesa sea resuelta, de manera correcta vamos a recibir una función con la respuesta de la petición.

En el siguiente ejemplo simulamos una request que nos devuelve un json: Parseamos la respuesta utilizando el método response.json() que también devuelve una promesa la cual podemos encadenar para finalmente obtener los nuestros datos.

Haciendo uso de la web jsonplaceholder podemos practicar estos conceptos:

```
Fetch('https://jsonplaceholder.typicode.com/users')
.then(response => response.json())
.then(json => console.log(json))
```

En la respuesta tenemos métodos que nos permiten interactuar con diferentes tipos de datos:

- arrayBuffer()
- blob()
- json()
- text()
- formData()
- Async/Await

Existe una alternativa al uso de promesas dentro de la método fetch y al encadenamiento del .then() que puede llevar a que tengamos muchas devoluciones de llamada anidadas, las cuales reducen rápidamente la legibilidad y pueden conducir fácilmente a un mal rendimiento o errores.

El uso de Async / Await no es totalmente compatible con todos los navegadores, por lo que debe ser consciente de esto y verificar sus necesidades al desarrollarlo. Aquí hay un recurso para verificar el soporte y la funcionalidad del navegador, y aquí para usar fetch .

Para llamar a una función usando la palabra clave await, la misma debe estar dentro de una función async como en el ejemplo a continuación.

```
async function getUser(name)
{
  let response = await      fetch(`https://api.github.com/users/${name}`);
  let data = await response.json()
  return data;
}
```

Enlaces:

https://www.youtube.com/watch?v=-j50jOzTHcI