AJAX

A principios del 2000, los/as desarrolladores/as web se encontraban ante el constante desafío de obtener o enviar información sin afectar el estado actual de la página, esto es, sin requerir una recarga completa (refresco de la página).

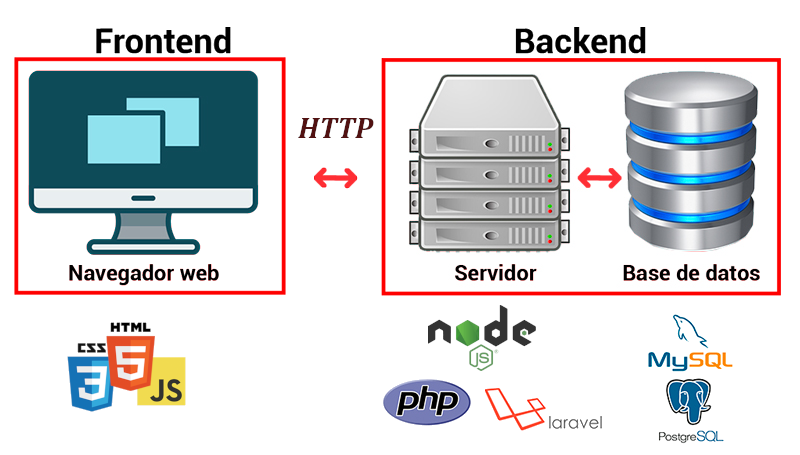
Si bien existían recursos como la etiqueta [<iframe>](https://developer.mozilla.org/es/docs/Web/HTML/Element/iframe) para cargar contenido anidado en el sitio, es decir, otro HTML dentro de la página actual, la necesidad de comunicar al cliente con el servidor, sin interrumpir el funcionamiento de la aplicación, derivó en la búsqueda de nuevas técnicas de programación.

Para dar respuesta a este problema, surgió [AJAX](https://developer.mozilla.org/es/docs/Web/Guide/AJAX/Getting_Started) (JavaScript Asíncrono y [XML](https://developer.mozilla.org/es/docs/Web/XML/XML_introduction)), un conjunto de técnicas de desarrollo que permiten que las aplicaciones web funcionen de forma asincrónica, pudiendo procesar tareas en segundo plano.

En consecuencia, cualquier app o web que emplee AJAX puede enviar y recibir datos sin volver a cargar toda la página, evitando la interrupción de acciones realizadas por el usuario, añadiendo interactividad y dinamismo a nuestra aplicación, **características esenciales del software moderno**.

**MODELO CLIENTE-SERVIDOR**

En primer lugar, repasemos el funcionamiento de una apliación web: como vimos en la primera clase, a este tipo de aplicaciones se accede teniendo conexión a internet, empleando un navegador y referenciando la dirección web del sitio (también llamado dominio, por ejemplo https://www.coderhouse.com/). El dominio está asociado a un [servidor](https://youtu.be/4Hcugh7z0cI), el cual no es más que un ordenador que tiene la aplicación web (se suele decir que el servidor aloja la aplicación web). Cuando el navegador del usuario carga el sitio, se puede visualizar la parte frontal de la aplicación (front-end o lado del cliente), con la que se puede interactuar.

x

Una aplicación moderna suele generar experiencias de usuario enriquecidas gracias a que se puede conectar a un conjunto de servicios de datos. Naturalmente, va a consumir numerosos recursos provistos por algún servidor (back-end), o enviar datos a éste para que se almacenen de forma persistente.

Como desarrolladores debemos aprender a dominar los métodos para realizar este intercambio de información y comprender el protocolo implicado.

**Peticiones HTTP**

HTTP (hypertext transfer protocol) es el mecanismo por el cual se piden y se proveen datos a través de internet. En nuestro caso el navegador, cuando se lo ordenemos, hará una *petición* (o *request*) *http* a algún servidor, éste la recibirá y la procesará, y nos devolverá en algún momento una respuesta con información que utilizaremos en la aplicación.

Estas peticiones que debemos hacer están definidas por varias partes:

* Una URL o dirección.
* Un *método* (GET, POST, PUT, DELETE).
* Headers
* Body
* Parámetros (*query params* o *url params*).

***url***

Cuando nos comunicamos con un servidor para pedir información lo hacemos a través de una url, ya que éste es un programa alojado en algún host y nos comunicamos con él a través de la dirección correcta.

***método***

Cada petición que hagamos viene acompañada por un *verbo*, que le indica al servidor cuál es nuestra intención con él. El servidor tiene la capacidad de escuchar distintas peticiones en la misma url, y decidir a cuál responder y cómo. Son 4 los verbos más utilizados, aunque hay muchos más:

* **GET**: una petición de tipo *get* se hace cuando quiero obtener información (o *recurso*) del servidor. Suelen ser las más utilizadas.
* **POST**: se utiliza cuando quiero enviar información al servidor para crear algún recurso.
* **PUT**: se utiliza para crear o modificar algún recurso en el servidor.
* **DELETE**: se utiliza para eliminar algún recurso en el servidor.

Las peticiones de tipo *POST* y *PUT* van acompañadas de un **body**, o cuerpo de la request, donde se definen los datos (información) a enviar al servidor. Las peticiones *GET* o *DELETE* no tienen body ya que no necesitan enviar datos adjuntos.

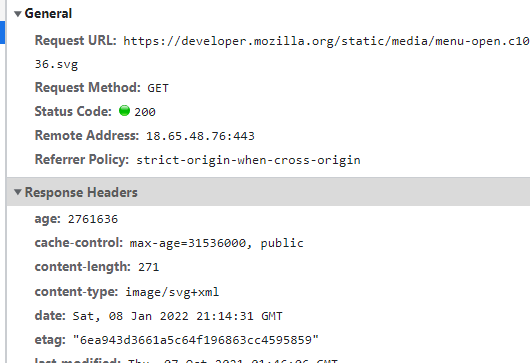
***body***

Es el espacio en la petición donde se definen los datos a enviar al servidor.



***headers***

Las cabeceras (headers) HTTP permiten al cliente y servidor enviar información sobre la petición y la respuesta. Los headers incluyen información sobre la petición para establecer una transferencia segura y clara, y de ser necesario se pueden modificar para agregar datos adicionales. No hay que confundir información sobre la *petición* (headers) con los datos que la petición puede transferir (body).



***parámetros***

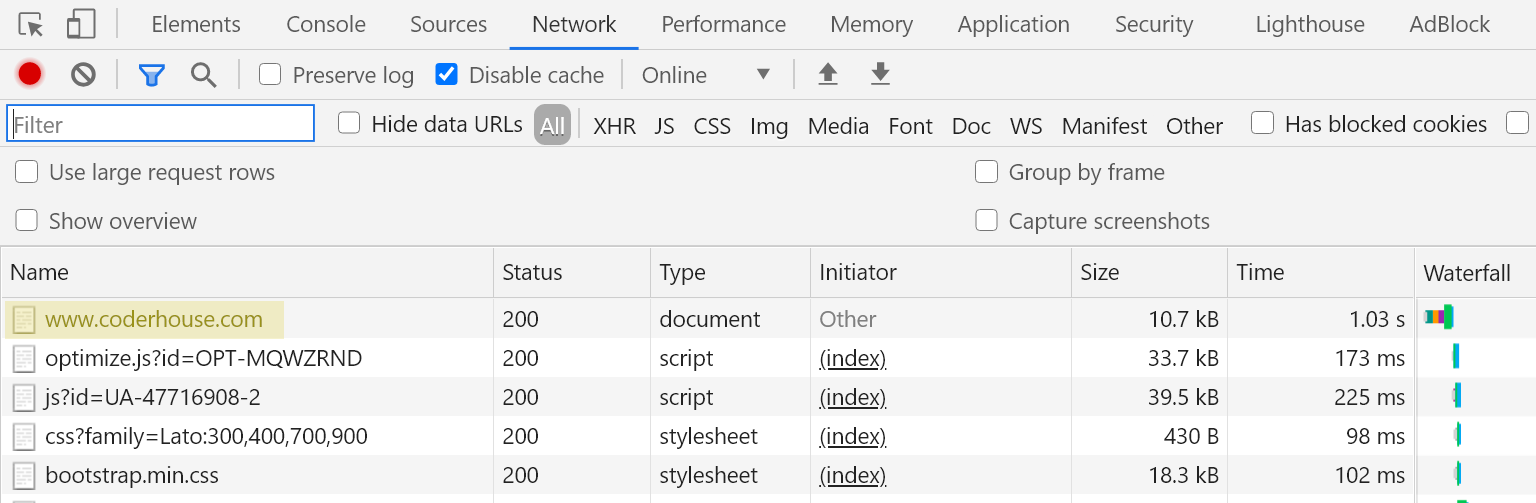
Para especificar una petición, se puede enviar información adicional en la forma de parámetros a través de la *url* . Tenemos dos formas de definir parámetros a través de la url:

* ***query params:*** esta sintaxis permite adjuntar en la url una serie de parámetros en la forma de pares *clave-valor*. Por ejemplo, si queremos buscar algo por google, debemos enviarle un valor de búsqueda por el parámetro *q*, a través de la url:.   
    
  Se utiliza el símbolo **?** para indicar el final de la parte de la *dirección* de la url y el comienzo del *query*. A partir de allí, se escriben parámetros con la forma *clave=valor* , pudiendo definir varios separándolos con el signo ampersand (&).  
  Por ejemplo, en la siguiente URL se hace una consulta a la PokeApi (<https://pokeapi.co/docs/v2>), pidiendo información al endpoint de */pokemon*, y se envían los parámetros *offset=0* y *limit=20* :  
  [https://pokeapi.co/api/v2/pokemon**?**offset=0**&**limit=20](https://pokeapi.co/api/v2/pokemon?offset=0&limit=20)  
  Ésto condiciona la búsqueda que queremos hacer en ese servidor.
* ***url params*:** esta sintaxis permite enviar parámetros directamente en la forma de segmentos de la url, es decir separados por **/** . Por ejemplo, la PokeApi nos indica lo siguiente:

https://pokeapi.co/api/v2/pokemon/{id or name}/  
Significa que ese {id or name} es un parámetro, un valor dinámico que insertamos en la url, en este caso para obtener información sobre un *pokemon* según su ID o nombre. Para obtener aquel con id = 1, haríamos una petición GET a la siguiente url: https://pokeapi.co/api/v2/pokemon**/1**

**Estados de la petición**

En la pestaña Network de las herramientas de desarrollador podemos ver todas las peticiones realizadas por el navegador. Vemos que, por ejemplo, al solicitar acceso a algún sitio web, el navegador realiza numerosas peticiones tipo GET para obtener recursos del servidor en cuestión. Estas peticiones proporcionan todos los archivos necesarios para montar la página en el browser (html, css, scripts, imágenes, etc.) y con eso poder visualizar la aplicación correctamente.



Cada petición tiene un *Status*, que es un código que significa el *estado* de la petición.

* Códigos de estado que comienzan en 2 (generalmente 200), significa que la petición fue exitosa.
* Los códigos que empiezan con 4 significan que hubo algún error con la petición. El famoso 404 indica que el recurso buscado no fue encontrado.
* Códigos que empiezan con 5 significan que hubo un error con el *servidor*, por lo tanto no es culpa de la petición.

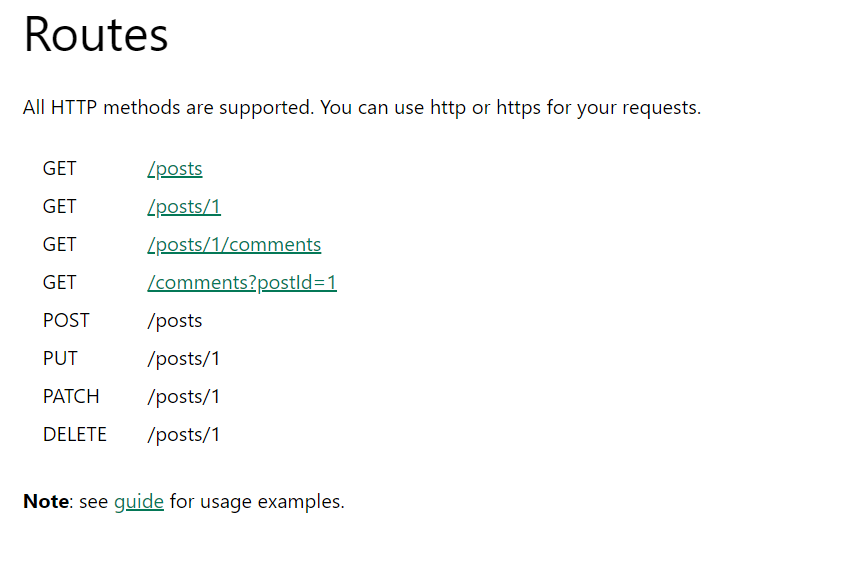
**API**

Una API (Application Programming Interfaces) es una aplicación web construida en base a la arquitectura [API REST](https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional), a la cual podemos solicitar y enviar información desde el cliente. Generalmente nos comunicamos con aplicaciones de este tipo y es la tendencia actual de desarrollo.

La ventaja de este modelo es que está orientado a recursos, y define métodos claros para solicitar y enviar información.

Hay muchas APIs disponibles que podemos utilizar para acceder a distintos recursos útiles para nuestra aplicación: servicios de contenido (CMS), plataformas de pago, servicios de email, etc. Incluso hay APIs creadas como bancos de información sobre series y videojuegos populares, como la PokeApi ([Documentation - PokéAPI](https://pokeapi.co/docs/v2)) o Star Wars API ([SWAPI](https://swapi.dev)).

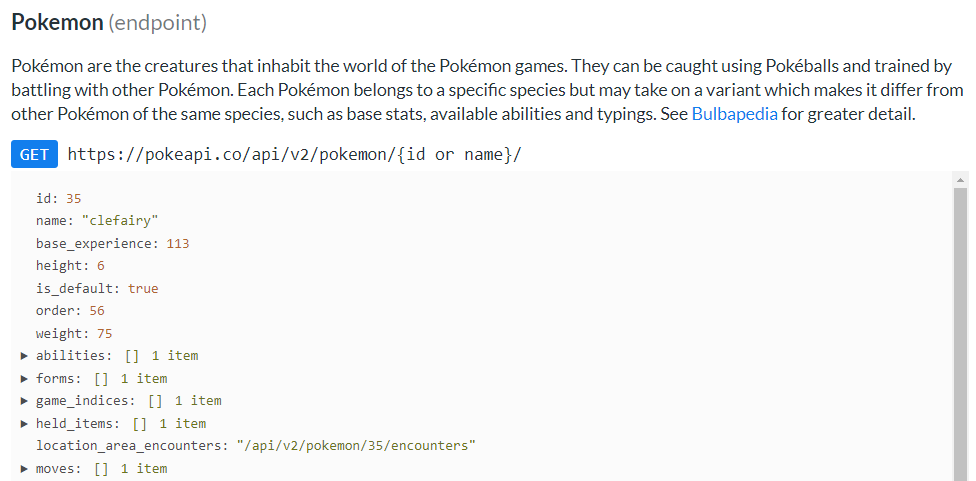
Si tomamos por ejemplo la API de [JSONPlaceholder](https://jsonplaceholder.typicode.com/), que sirve para hacer pruebas de peticiones, podremos notar que existen distintas rutas que podemos usar:



Una API suele tener una *url base*, que es el dominio donde está alojada la aplicación, y luego puede tener varios *endpoints*, que son distintas secciones a las que podemos acceder. Hacer peticiones a distintos endpoints nos proporcionará distintos recursos alojados en ese backend. A la vez, se pueden hacer peticiones con distintos métodos al mismo endpoint y obtener distintos resultados.

Generalmente, similar a cuando queremos incorporar una librería, al momento de consumir una API debemos revisar su *documentación*, ya que ésta nos define los distintos *endpoints* disponibles, qué métodos utilizar para hacer una petición y qué nos ofrecerá en respuesta.

Por ejemplo, la Poke Api nos ofrece la opción de consultar sobre algún pokemon. Nos dice qué método utilizar en qué url, y un ejemplo del tipo de respuesta que tendremos si la petición es exitosa (<https://pokeapi.co/docs/v2#pokemon>):



FETCH

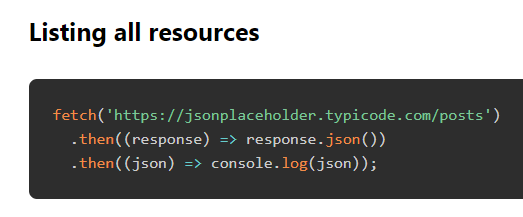
Javascript nos ofrece el método *fetch()* para hacer peticiones HTTP a algún servicio externo. Como éstas peticiones son asincrónicas, convenientemente el método *fetch()* trabaja con **promesas**.

El método recibe un primer parámetro que es la *url* a la cual hacer la petición, y un segundo parámetro opcional de configuración:

| fetch(url, config) |
| --- |

Para los siguientes ejemplos utilizaremos la API de JSON Placeholder, diseñada para hacer pruebas de peticiones ([JSONPlaceholder - Guide](https://jsonplaceholder.typicode.com/guide/) ), simulando un listado de *posts*.

Por defecto el método fetch hace peticiones del tipo GET. Según la documentación, para obtener una lista de posts debemos hacer una petición del siguiente tipo:



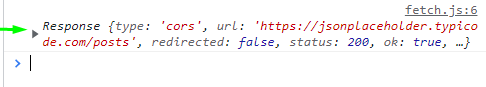
Vayamos por parte para entender este proceso.

Primero, llamemos al método con la url correspondiente y veamos qué retorna:

| console.log( fetch('https://jsonplaceholder.typicode.com/posts') )  // Promise {<pending>} |
| --- |

Ésta retorna una Promesa pendiente. Para trabajar con la resolución de la petición, debemos hacerlo dentro del *.then()* correspondiente:

| fetch('https://jsonplaceholder.typicode.com/posts')  .then( (resp) => console.log(resp) ) |
| --- |



Haciendo console.log de la respuesta, no vemos el listado de posts que esperamos sino un objeto del tipo Response.

**Response**

Llamar a *fetch()* retorna una promesa que resuelve en un **objeto Response** que contiene información sobre la respuesta del servidor, como su código de estado y headers.

Para acceder al **contenido** de la respuesta debemos dar un paso adicional, y por eso es que se ven dos *.then()* concatenados.

Generalmente se *transfieren datos en formato JSON.* Por lo tanto, para obtener el contenido de la respuesta debemos aplicar el método *.json()* a ese objeto. Éste retorna a su vez una Promesa, por eso es que capturamos su contenido (los datos enviados por la API) en un segundo .then() :

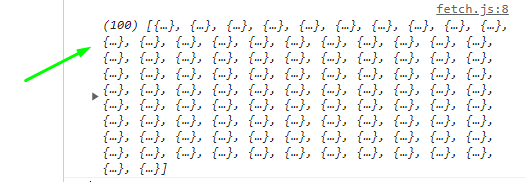
fetch('https://jsonplaceholder.typicode.com/posts')

.then( (resp) => resp.json() )

.then( (data) => {

console.log(data)

})

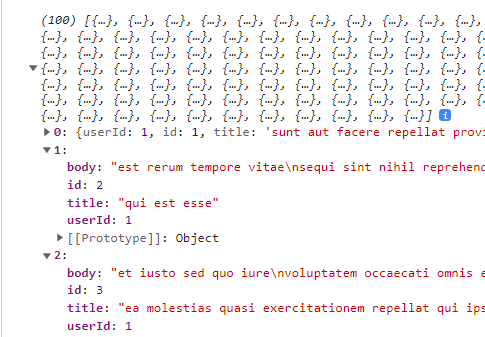


En el parámetro *data* tenemos el contenido de la respuesta de nuestra petición. En este caso, la API nos responde con un array de 100 elementos, donde cada elemento es un *post*.

**Analizando respuestas**

Trabajar con APIs nos ofrece un entorno claro sobre cómo comunicarnos y obtener respuestas con recursos. Sin embargo, cada API define qué responder, qué formato darle a los datos que envía y cómo estructurarlos. Por lo tanto, siempre debemos *analizar* las respuestas obtenidas para ver qué datos utilizar de ellas.

En este caso, veamos cómo son los objetos del array anterior:



Cada elemento tiene propiedades *body, id, title, userId*. Recordemos que estamos trabajando con el parámetro *data* definido, que es un array de objetos. Por lo que podemos recorrerlo y acceder a sus objetos y propiedades:

fetch('https://jsonplaceholder.typicode.com/posts')

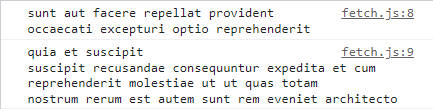
.then( (resp) => resp.json() )

.then( (data) => {

console.log( data[0].title )

console.log( data[0].body )

})

Teniendo esto disponible dentro del *.then()*, podemos volcarlo al DOM utilizando los métodos vistos previamente!

<h2>Posts!</h2>

<hr/>

<ul id="listado">

</ul>

// JS

const lista = document.querySelector('#listado')

fetch('https://jsonplaceholder.typicode.com/posts')

.then( (resp) => resp.json() )

.then( (data) => {

data.forEach((post) => {

const li = document.createElement('li')

li.innerHTML = `

<h4>${post.title}</h4>

<p>${post.body}</p>

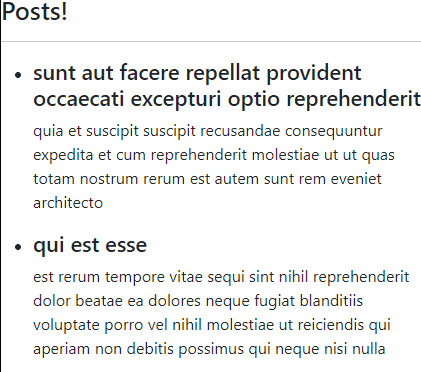
`

lista.append(li)

})

})

Al obtener la respuesta de la API, recorremos el array obtenido y agregamos a la <ul> un elemento <li> con el contenido de cada post en el array. Vemos en la página un listado de contenido obtenido desde un servicio externo:



**Enviando datos con POST**

La api de JSON Placeholder también nos permite simular peticiones POST. Es decir que podemos hacer una petición para *enviar datos* a la API. Al ser una simulación no se crean recursos realmente en el servidor, pero sí obtenemos una respuesta aceptando el POST.

Dijimos que el segundo parámetro del método fetch es un objeto de configuración. En éste podemos definir el método, los headers y el body de la petición. Si bien fetch trae valores por defecto para ésto (como el método que es GET por defecto), podemos modificarlo a discreción según sea necesario.

En este caso la documentación nos indica que para hacer un post debemos hacer un *fetch* con las siguientes características:

fetch('https://jsonplaceholder.typicode.com/posts', {

method: 'POST',

body: JSON.stringify({

title: 'Coderhouse',

body: 'Post de prueba',

userId: 1,

}),

headers: {

'Content-type': 'application/json; charset=UTF-8',

},

})

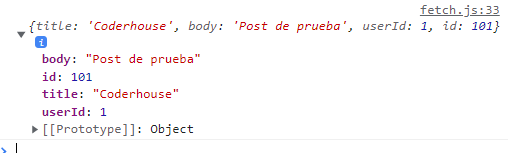
.then((response) => response.json())

.then((data) => console.log(data))

En el objeto de configuración tenemos varias propiedades a definir:

* *method:* *‘POST’*. Significa que el método de la petición será POST. Si no lo modificamos será de tipo GET por defecto.
* *headers:* en este caso se agrega una propiedad ‘Content-type’, con el valor que nos indica la documentación de la API. Si no se agrega la petición sería rechazada por el servidor.
* *body:* aquí se adjuntan los datos a enviar al servidor. En este caso se envía un objeto con la forma *{ title, body, userId }*. El body debe enviarse en formato JSON, por eso lo vemos envuelto en un JSON.stringify().

Por lo general al hacer un POST obtenemos una respuesta que nos envía una copia del recurso creado en el servidor. La forma de trabajar la respuesta es la misma que la anterior. :



**Rutas relativas**

Si la url utilizada no contiene el prefijo ‘https:’, se entiende que la ruta es *relativa*. Por lo tanto, podemos hacer una petición a algún archivo local en formato JSON utilizando fetch.

Por ejemplo, creemos un archivo *data.json* que simule un array de productos:

// data.json

[

{"nombre": "Producto 1", "precio": 1500, "id": 1},

{"nombre": "Producto 2", "precio": 2500, "id": 2},

{"nombre": "Producto 3", "precio": 3500, "id": 3},

{"nombre": "Producto 4", "precio": 4500, "id": 4},

{"nombre": "Producto 5", "precio": 5500, "id": 5}

]

Nótese que debe estar escrito con el formato *json* válido.

Ahora al momento de cargar la aplicación, podemos llamar a este archivo con *fetch* y generar una vista de forma asincrónica:

const lista = document.querySelector('#listado')

fetch('/data.json')

.then( (res) => res.json())

.then( (data) => {

data.forEach((producto) => {

const li = document.createElement('li')

li.innerHTML = `

<h4>${producto.nombre}</h4>

<p>${producto.precio}</p>

<p>Código: ${producto.id}</p>

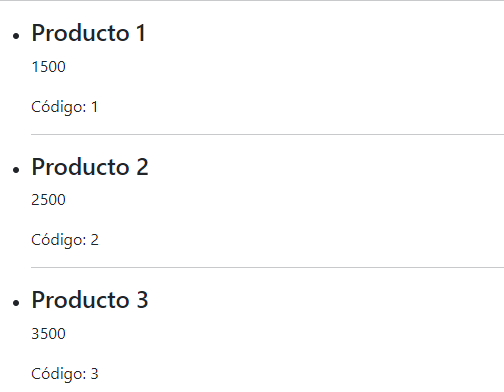
<hr/>

`

lista.append(li)

})

})



Al ser un archivo local la respuesta es casi inmediata, pero sigue siendo un proceso asincrónico.

**ASYNC - AWAIT**

Trabajar con promesas facilita mucho el control de los procesos asincrónicos. Sin embargo, en procesos extensos se puede dificultar el trabajo escribiendo todo dentro de varios *.then()*.

Por suerte los desarrolladores de JS ya pensaron en esto y nos ofrecen una herramienta que nos permite trabajar las promesas como si escribiéramos código sincrónico: *async await*.

El método *fetch* **retorna** una promesa. De forma sincrónica, si guardamos esta promesa en una variable veremos la promesa pendiente, porque esto sucede sincrónicamente:

const resp = fetch('https://jsonplaceholder.typicode.com/posts')

console.log(resp) // Promise {<pending>}

Significa que el console.log() *no espera* a que se resuelva la promesa de la línea anterior para ejecutarse.

La sentencia **await** nos permite establecer un punto de espera en el código. Aplicado como prefijo a una promesa (en este caso, el return del fetch) se *bloquea* la ejecución de la siguiente instrucción hasta que la promesa se vea resuelta. Así, agregando esta sentencia podemos ver que ahora en la variable vemos el objeto Response, o sea la promesa resuelta.

const resp = await fetch('https://jsonplaceholder.typicode.com/posts')

console.log(resp) // Response

Pero esta sentencia **await** sólo puede utilizarse dentro de una **función asincrónica**. Aquí es donde entra la sentencia **async**. Ésta palabra reservada sirve para declarar una función como asincrónica, y se agrega como prefijo a la función:

async function pedirPosts() { }

// o bien

const pedirPosts = async () => { }

Ahora bien, dentro de una función *async* podemos utilizar la sentencia *await* vista previamente. Esto nos permite *esperar* a que se resuelvan las promesas vistas para continuar con la instrucción siguiente. El resultado es una sintaxis que se asemeja a la escritura sincrónica tradicional:

| const pedirPosts = async () => {  const resp = await fetch('https://jsonplaceholder.typicode.com/posts')  const data = await resp.json()    data.forEach((post) => {  const li = document.createElement('li')  li.innerHTML = `  <h4>${post.title}</h4>  <p>${post.body}</p>  `  lista.append(li)  })  }  pedirPosts() |
| --- |

Obtenemos el mismo resultado que antes, pero con una sintaxis más clara. El *async-await* funcionan de la mano, y es sólo una herramienta adicional que puede facilitar la escritura. No es una obligación.