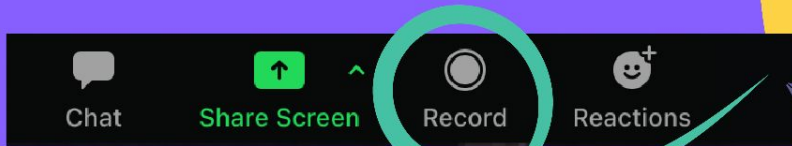




programee

***¡Hey!
No olvides
poner a **grabar**
la clase***



Objetivos de la clase



Objetivos

- Conocer sobre AJAX y el modelo cliente-servidor
- Comprender la estructura de una petición y sus métodos
- Entender qué es una API y cómo nos comunicamos con ellas
- Aprender a usar FETCH y vincular resultados con el DOM

Clase 14. JAVASCRIPT

AJAX



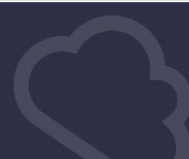
En las primeras etapas del desarrollo del desarrollo web, algunos de los grandes problemas que había era que cuando queríamos mostrar un cambio en un sitio sin importar lo pequeño que sea teníamos que crear un archivo completamente nuevo exactamente igual al anterior solo con este cambio para que el usuario con un click refresque toda la página, generando así un tiempo de espera porque todo el documento se carga de nuevo.

Por otro lado cuando se trataba de evitar esto mostrando el contenido en una sola página sin actualizar el navegador y, ocultando el contenido para luego mostrarlo según la interacción, se tenía que cargar todo en el mismo archivo, generando archivos super pesados y con gran tiempo de carga. Para solucionar esto se empezaron a desarrollar técnicas para la carga asíncrona de contenidos en una página, es decir métodos para actualizar sectores de nuestros sitios usando archivos externos.

Una de las primeras tecnologías introducidas fueron iframe (introducido en 1996) y el tipo de elemento layer (introducido en 1997, actualmente abandonado su desarrollo). Estos contaban con el atributo src que podía tomar cualquier dirección URL externa, y cargando una página que contenga JavaScript.

AJAX fue creado en 2005 por Jesse James Garrett y surgió como una alternativa más elegante para estas técnicas, permitiéndonos solicitar al servidor nuestros contenidos, cargándose en segundo plano sin interferir con la visualización ni la operatividad de la página.

AJAX es el acrónimo de **Asynchronous JavaScript And XML** (JavaScript asíncrono y XML) es decir, es un término que describe un modo de utilizar conjuntamente varias tecnologías. Esto incluye: HTML o XHTML, CSS, JavaScript, DOM, XML, XSLT, y lo más importante, el objeto XMLHttpRequest.



MODELO CLIENTE-SERVIDOR

La arquitectura cliente-servidor es un modelo de diseño de software el cual representa la forma en la que se producen las comunicaciones entre dos nodos de una red, uno de los nodos que forma parte de la comunicación tiene el rol de cliente, y otro nodo tiene el rol de servidor.

Los componentes que conforman este modelo son:

Cliente: En este caso el cliente es quien solicita la información proveniente de la red, este puede ser un ordenador, teléfono o inclusive una aplicación informática.

Servidor: Es aquel que provee los servicios, es quien envía la información a los demás agentes de la red.

Protocolo: Un protocolo es un conjunto de normas o reglas y pasos establecidos de manera clara y concreta, rigen tanto el formato como el control de la interacción entre los distintos dispositivos dentro de una red o sistema de comunicación, cuyo objetivo es que puedan transmitir datos entre ellos

Red: Denominamos red al conjunto de clientes, servidores y base de datos unidos de una manera física o no física en el que existen protocolos de transmisión de información establecidos.

Servicios: Definimos como Servicios al conjunto de información que busca responder las necesidades de un cliente, donde esta información pueden ser mail, música, mensajes simples entre software, videos, etc.

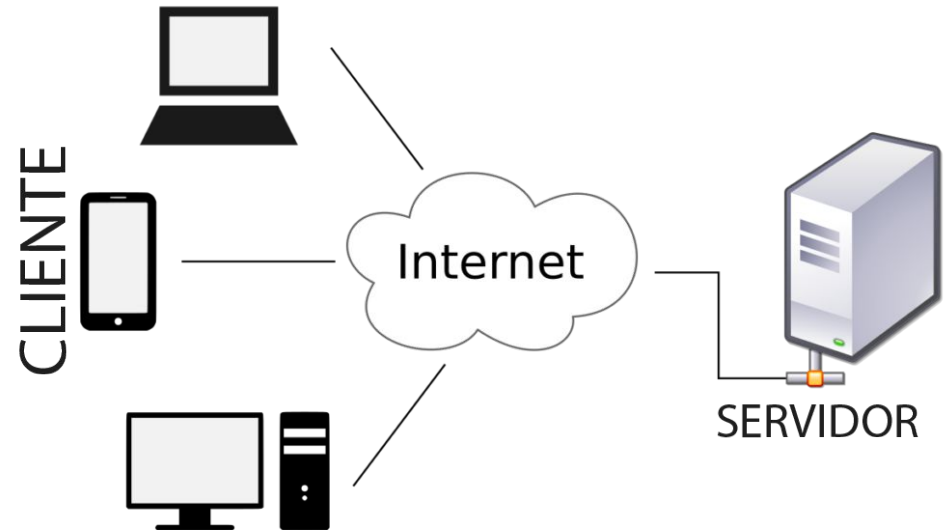
Base de datos: Una base de datos es un almacén de información o datos estructurados, perteneciente a un mismo contexto, la cual está ordenada de modo sistemático para facilitar su posterior recuperación, análisis y/o transmisión.



MODELO CLIENTE-SERVIDOR

El funcionamiento de esta arquitectura se basa principalmente en la aplicación de diferentes modelos informáticos para que al enviar una solicitud se pueda acceder a un servicio de un servidor, esta comunicación se la hace a través de una red mediante protocolos establecidos y el correcto almacenamiento de la información.

La arquitectura cliente servidor más conocida es la “red de Internet” donde se conectan tanto ordenadores teléfonos y otros dispositivos de diferentes personas conectadas alrededor del mundo, estas a su vez se conectan a los servidores de su proveedor de Internet por ISP donde son redirigidos a los servidores de las páginas que contienen la información que se desea visualizar, de esta forma permitimos la distribución de la información en forma ágil y eficaz.



PETICIONES HTTP

Cuando un usuario navega por internet, el navegador actúa como cliente el cual se comunica con los distintos servidores web con la ayuda del protocolo HTTP.

Cuando un cliente hace una petición a un servidor lo que hace básicamente es enviar un mensaje al servidor usando el protocolo HTTP la que el servidor procesa y retorna una respuesta usando el mismo protocolo, a estos dos procesos se los llaman peticiones y respuestas HTTP

Un HTTP request se compone de:

Path

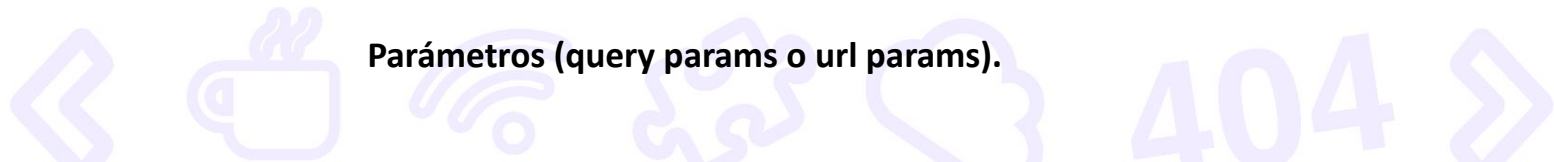
Método

Protocolo

Headers.

Body

Parámetros (query params o url params).



PATH

Para poder comunicarnos con un servidor utilizamos la URL, es decir la dirección en donde se encuentra el recurso

Métodos

Los Métodos HTTP o también llamados HTTP verbs, indican el formato de comunicación entre el cliente y servidor web, Es decir indica qué tipo de request es.

Los más utilizados son:

Método GET: Estas peticiones se utilizan cuando queremos recuperar o extraer información del servidor.

Método PUT: Usamos este método para agregar o modificar un recurso en el servidor.

Método POST: Usamos este método para enviar información al servidor mas no para agregar o modificar un nuevo elemento, este es principalmente usado para el envío de los formularios.

Método DELETE: Con este método lo que hacemos es solicitarle al servidor que borre uno o varios recursos.



Otros métodos

Método HEAD: Este método lo utilizamos cuando no queremos recibir el archivo completo, es decir solo solicitamos su encabezado.

Método OPTIONS: Con este lo que hacemos es solicitar al servidor que nos informe qué métodos soporta.

Método TRACE: Este método es principalmente usado para el diagnóstico en la conexión con los servidores, ya que permite monitorear los mensajes que hay entre el cliente y el servidor web, es decir con esto obtenemos la ruta que sigue una HTTP request durante todo su camino (desde que se realiza, hasta que llega al servidor y vuelve de regreso al cliente)

Método CONNECT: Este método se utiliza para solicitar una conexión de tipo túnel TCP/IP. Principalmente se utiliza cuando se necesita utilizar un proxy para una conexión segura cifrada HTTPS o para comunicaciones vía SSL.



Protocolo

Contiene HTTP y su versión. Con el http se establecen criterios de sintaxis y semántica informática.

Headers

Las cabeceras HTTP son la parte central de esas solicitudes y respuestas HTTP, tienen un esquema de key: value la cual contienen información sobre el navegador cliente, la página solicitada, el servidor, el HTTP request, etc.

Body

Aquí es donde definimos la información que se envía al servidor a través de POST o PUT.

Web Server

Client



Google
www.myserver.com/sales/item_price.dll

Item

Brand

HTTP Request

HTTP

Status Line	POST /sales/item_price.dll HTTP/1.0
Header	From: Mike User-Agent: Mozilla/5.0 (Windows NT 6.3) Content-Type: application/x-www-form-urlencoded Content-Length: 36
Body	item=hard+drive&brand=sony&Find=Find

HTTP Response

Status Line	HTTP/1.0 200 OK
Header	Date: Tue, 31 Dec 2019 20:59:59 GMT Content-Type: text/html; charset=utf-8 Content-Length: 224
Body	<html> <head> <title>Item Information</title> </head> <body> <h1>Sony Hard Drives</h1> <table> <tr> <td>Hard drive Sony SN-3</td> <td>\$150.00</td> </tr> </table> </body> </html>

HTTP

Parámetros

Parámetros (query params o url params).

Podemos especificar aún más la petición que queremos realizar al servidor enviando "parámetros", esta no es más que una información adicional que agregamos a la url.

Query Params: con esta sintaxis utilizamos el símbolo ? para separar la url de los parámetros y a su vez indicar el comienzo de los mismos, cada parámetro se lo coloca con el formato de clave valor, separando la clave y el valor con el símbolo =, y a su vez usando como separador entre los parámetros al símbolo &

<https://www.google.com/search?q=pizza>

URL params: Esta es una sintaxis visualmente más limpia que la anterior ya que nos permite enviar los parametros como un segmento de la url es decir separado por / es decir, en lugar de colocar:

<https://www.disneyplus.com/es-419?brand=mavel>

Colocaremos: <https://www.disneyplus.com/es-419/brand/mavel>



Estados de la petición

Cada vez que el cliente manda una petición el servidor la procesa y devuelve dos cosas, los datos que se pidieron y un código de estado.

Todas las peticiones de nuestro navegador las podemos ver en la sección Network de nuestro inspector de elementos

Cada elemento de la **tabla Status** contiene un código y cada uno de estos significa el estado de la petición..

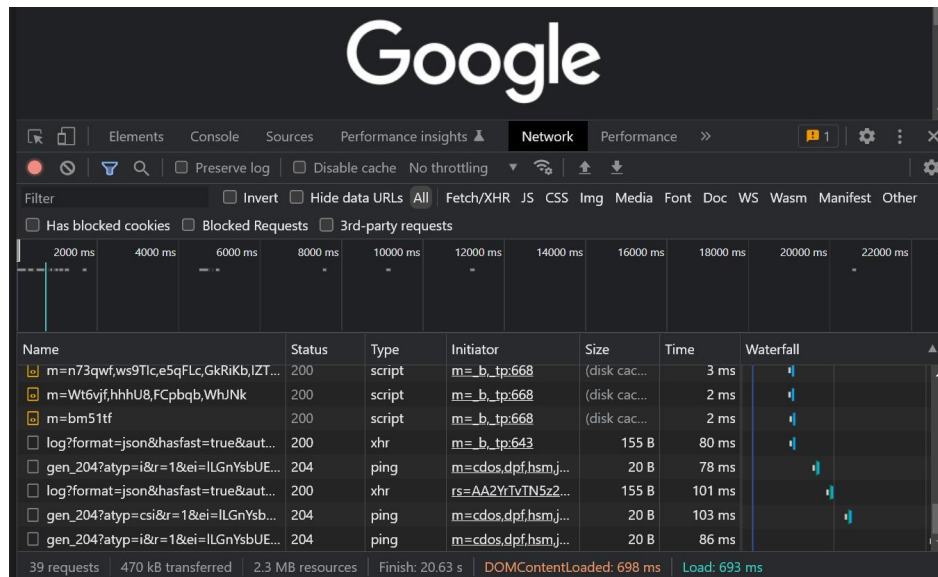
100s: Códigos informativos que indican que la solicitud iniciada por el navegador continúa.

200s: Estos son códigos que devuelven el éxito de una operación es decir cuando la solicitud del navegador fue recibida, entendida y procesada por el servidor.

300s: Códigos de redireccionamiento devueltos cuando un nuevo recurso ha sido sustituido por el recurso solicitado.

400s: Estos se generan para indicar al cliente que que hubo un problema con la solicitud. El famoso 404 indica que el recurso buscado no fue encontrado.

500s: al encontrarnos con estos nos indican que la solicitud fue aceptada, pero que un error en el servidor impidió que se cumpliera.



API

API

Una API, o interfaz de programación de aplicaciones, es un conjunto de reglas que permite que diferentes programas se comuniquen entre sí, permitiendo de esta manera una forma flexible y ligera de solicitar y enviar información por parte del cliente, y además surgiendo como el método más común para conectar componentes en la arquitectura de microservicios.

Actualmente existen muchas APIs tanto pagas como gratuitas que podemos usar para consumir directamente en nuestros proyectos o simplemente para realizar pruebas con ellas.

Podemos encontrar algunas de ellas buscándolas en google en el repositorio de git

Este repositorio cuenta con una gran cantidad de apis gratuitas y está organizado con un índice para que seleccionemos la Api de la categoría que prefiramos.

Google

github public apis



All

Videos

News

Images

Maps

More

Tools

About 79,300,000 results (0.34 seconds)

<https://github.com> > [public-apis](#) > [public-apis](#)

[public-apis/public-apis: A collective list of free APIs - GitHub](#)

A collective list of free **APIs**. Contribute to **public-apis/public-apis** development by creating an account on **GitHub**.

[Public-apis](#) · [README.md](#) · [Issues 3](#) · [Pull requests](#)



GitHub - public-apis/public-apis



github.com/public-apis/public-apis



README.md

Index

- [Animals](#)
- [Anime](#)
- [Anti-Malware](#)
- [Art & Design](#)
- [Authentication & Authorization](#)
- [Blockchain](#)
- [Books](#)
- [Business](#)
- [Calendar](#)
- [Cloud Storage & File Sharing](#)
- [Continuous Integration](#)
- [Cryptocurrency](#)
- [Currency Exchange](#)
- [Data Validation](#)



ROUTES

Para poder acceder a las APIS solemos hacerlo por medio de una URL base y a su vez a partir de esta obtener varios endpoints que nos permiten hacer una selección un poco más fina de qué es lo que queremos que nos retorne esta API.

Usualmente cuando buscamos Apis para consumir estas, al igual que las librerías vienen respaldadas con una documentación de uso, la cual nos ayudará a hacer las peticiones adecuadamente.

Como ejemplo podemos usar la api de Songsterr esta api nos proporciona un listado de canciones el cual podemos filtrar el resultado para que nos muestre solo las canciones de un artista o grupo específico:

`https://www.songsterr.com/a/ra/songs/byartists.xml?artists=versailles`

o también por palabras claves:

`https://www.songsterr.com/a/ra/songs.xml?pattern=pizza`



FETCH

Es un método moderno y versátil para enviar peticiones a algún servicio externo y **consumir estos recursos de forma asíncrona**.

La ventaja principal de **Fetch** es que funciona con promesas no con devoluciones de llamadas, esto es mejor ya que facilita la escritura y el manejo de las solicitudes asíncronas.



Donde en **url** colocaremos la dirección URL a la que deseamos acceder y en **options** los parámetros opcionales, como puede ser un método o los encabezados de nuestra petición, etc.

FETCH

Por ejemplo utilizaremos la API que proporciona datos.gob.es la cual nos proporciona algunas url útiles de la zona que le indiquemos.

```
fetch('https://datos.gob.es/apidata/nti/territory/Province/Madrid')  
  .then( response => console.log(response))
```

Siempre es mejor colocar todas las operaciones dentro de .then ya que si lo hacemos al revés nos retornará una promesa pendiente.

```
console.log(fetch('https://datos.gob.es/apidata/nti/territory/Province/Madrid') );  
// Promise {<pending>}
```



FETCH

Response

Cada vez que usamos **Fetch** para hacer nuestras peticiones, este toma como argumento la ruta del recurso y por resultante nos devuelve un objeto promise el cual contiene la respuesta, es decir un objeto response.

El formato que usualmente se retornan los datos es JSON por lo que para poder utilizarlos tenemos que aplicar el método `json()`, es decir dentro de `.then` en donde indicamos las acciones a seguir para nuestro response, antes de operar con esta respuesta allí es donde tenemos que aplicar este método

```
fetch('https://datos.gob.es/apidata/nti/territory/Province/Madrid')  
  .then( response => {...})
```



FETCH

Analizando respuestas.

El gran beneficio de trabajar con apis es que podemos operar con recursos que pertenezcan tanto a nuestro servidor como externos, dándonos una gran facilidad a la hora de manipular nuestros distintos proyectos.

Supongamos que utilizamos la API de randomuser.com para obtener nuestra respuesta

```
{
  "results": [
    {
      "gender": "male",
      "name": {
        "title": "Monsieur",
        "first": "Josef",
        "last": "Roussel"
      },
      "location": {
        "street": {
          "number": 1677,
          "name": "Place de la Mairie"
        },
        "city": "Ponte Capriasca",
        "state": "Genève",
        "country": "Switzerland",
        "postcode": 4482,
        "coordinates": {
          "latitude": "-50.3475",
          "longitude": "-162.0455"
        },
        "timezone": {
          "offset": "-1:00",
          "description": "Azores, Cape Verde Islands"
        }
      },
      "email": "josef.roussel@example.com",
      "login": {
        "uuid": "8e624a4c-d158-4ea9-95d5-10a6677573a3",
        "username": "smallfrog158",
        "password": "english"
      },
      "phone": "077 434 16 27",
      "cell": "078 786 22 41",
      "id": {
        "name": "AVS",
        "value": "756.6724.8356.39"
      },
      "picture": {
        "large": "https://randomuser.me/api/portraits/men/41.jpg",
        "medium": "https://randomuser.me/api/portraits/med/men/41.jpg",
        "thumbnail": "https://randomuser.me/api/portraits/thumb/men/41.jpg"
      },
      "nat": "CH"
    }
  ],
  "info": {
    "seed": "2915b71872d787d6",
    "results": 1,
    "page": 1,
    "version": "1.3"
  }
}
```



FETCH

Al momento de recibirlo, y para poder trabajar con estos datos como un objeto, tenemos que aplicarle el método `json()`

Igualmente podemos recorrer todo el objeto y crear listados con formatos específicos usando este contenido

```
fetch('https://randomuser.me/api/?results=10')  
  .then( response => response.json())  
  .then( datta => console.log(datta["results"][0].name))
```

```
fetch('https://randomuser.me/api/?results=10')  
  .then( response => response.json())  
  .then( (data) => {  
    data.forEach((post) => {  
      const li = document.createElement('li')  
      li.innerHTML = '  
        <h4>  
          ${post["results"][0].name.first}  
          ${post["results"][0].name.last}  
        </h4>  
        <p>Genero: {post["results"][0].gender}</p>  
      '  
      lista.append(li)  
    })  
  })
```

FETCH

Enviando datos con POST

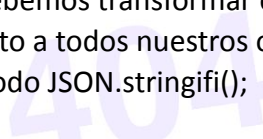
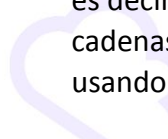
Existen casos en los que tendremos que simular peticiones POST al enviar datos a la API, podemos configurar esto en el segundo parámetro de nuestro método fetch.

```
fetch('https://randomuser.me/api/?results=10', {  
  method: 'POST',  
  body: JSON.stringify({  
    title: 'Programa',  
    body: 'Usuarios Random',  
    userId: 1,  
  }),  
  headers: {  
    'Content-type': 'application/json; charset=UTF-8',  
  },  
})
```

Method: como indica la palabra, en method colocaremos el método HTTP que se tendrá en la petición en este ejemplo será POST. Y en el caso de no colocar nada por defecto será de tipo GET.

Headers: este es el encabezado de nuestra petición, los datos que coloquemos aquí permiten que el navegador pueda controlar estas peticiones de manera correcta y segura para evitar ser rechazada por el servidor.

Body: Aquí colocaremos el cuerpo de nuestra respuesta, es decir los datos que enviaremos al servidor, estos datos deben enviarse como una cadena de texto JSON es decir que debemos transformar en cadenas de texto a todos nuestros objetos usando el método `JSON.stringify()`;

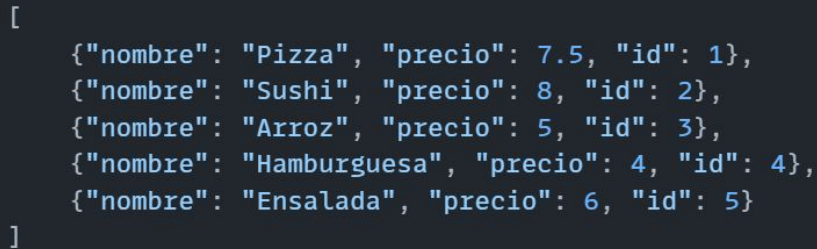


RUTAS RELATIVAS

Las rutas relativas son aquellas que a diferencia de las rutas absolutas, no contienen la URL completa del documento, es decir que al escribirlas estas estan sin el protocolo (http://) ni el nombre de dominio, sino solamente el nombre del archivo a referenciar y la carpeta en la que se encuentra.

Estas son convenientes cuando queremos referenciar un archivo local.

Por ejemplo si tenemos un archivo JSON como el siguiente:



```
[
  {"nombre": "Pizza", "precio": 7.5, "id": 1},
  {"nombre": "Sushi", "precio": 8, "id": 2},
  {"nombre": "Arroz", "precio": 5, "id": 3},
  {"nombre": "Hamburguesa", "precio": 4, "id": 4},
  {"nombre": "Ensalada", "precio": 6, "id": 5}
]
```


RUTAS RELATIVAS

Para poder cargarlo a nuestro documento con **Fetch** usamos la siguiente sintaxis.

```
let Mimenu = document.querySelector('#Menulist');

fetch('/menu.json')
  .then( (res) => res.json())
  .then( (data) => {

    data.forEach((comida) => {
      const li = document.createElement('li')
      li.innerHTML = `
        <h4>${comida.nombre}</h4>
        <p>${comida.precio}</p>
      `
      Mimenu.append(li)
    })
  })
```

Lo cual como resultado obtendremos algo como lo siguiente:

MENU

Pizza	\$7.5
Sushi	\$8
Arroz	\$5
Hamburguesa	\$4
Ensalada	\$6

En este caso aunque en la carga es un proceso asíncrono, la respuesta en sí de los archivos locales es prácticamente inmediata.

ASYNC - AWAIT

ASYNC - AWAIT

Existe una sintaxis que nos ayuda a trabajar más cómodos con nuestras promesas.

```
async function miFuncion() {  
  return "Pizza";  
}
```

Cuando colocamos la palabra `async` delante de una función convertimos esta función en asíncrona, y por consiguiente lo que devolverá será un elemento del tipo `promise`.

En el caso que utilicemos arrow function podemos trabajar con `async` de la siguiente manera

```
const MiMenu = async () => "Pizza";
```



ASYNC - AWAIT

Por otro lado, podemos utilizar **Await** para manejar esta promesa. Await causa una pausa en la ejecución de la función asíncrona para de esta manera ayudarnos a esperar el resultado de nuestra promesa

```
const MiMenu = async () => "Pizza";  
const valor1 = MiMenu(); // Promise {<fulfilled>: 'Pizza'}  
const valor2 = await MiMenu(); // 'Pizza'
```

Si al agregar **Await** en una función que transformamos en asincrónica con Async, esta "espera" haciendo que nuestra respuesta siempre sea una promesa que ha sido cumplida, ¿Podríamos hacer lo mismo con Fetch()?

La respuesta es sí, como lo que retorna Fetch es una promesa, al usar Await podemos establecer un punto de espera



ASYNC - AWAIT

```
console.log(fetch('https://datos.gob.es/apidata/nti/territory/Province/Madrid') );  
// Promise {<pending>}
```

En este el resultado de nuestra promesa tendrá el valor de “Pendiente” mientras que si usamos Await

```
let resp = await fetch('https://datos.gob.es/apidata/nti/territory/Province/Madrid') ;  
console.log(resp );  
//Promise {<fulfilled>: Response}
```



ASYNC - AWAIT

Un detalle importante a tener en cuenta es que await solo funciona dentro de una función asíncrona, dicho esto podemos ver otra forma de trabajar con await directamente dentro de las funciones asíncronas:

```
let ContenidoMenu;
let MiMenu = async () => {
  let BDmenu = await fetch('/menu.json');
  let listado = await BDmenu.json();
  listado.forEach((comida) => {
    const li = document.createElement('li')
    li.innerHTML = `
      <h4>${comida.nombre}</h4>
      <p>${comida.precio}</p>
    `
    ContenidoMenu.append(li)
  })
}
```



FIN

