

DWC

(Desarrollo Web en entorno cliente)



JavaScript

Tema 7.2

Comunicación Asíncrona

Índice

1.- Comunicación asíncrona en JavaScript	1
2.- AJAX.....	3
2.1- Funcionamiento del objeto XMLHttpRequest.....	4
2.2- Petición get.....	6
2.3- Petición delete.....	9
2.4- Enviar datos al servidor post, put y patch.....	10
3.- Promesas.....	12
3.1.- Then	14
3.2.- Catch.....	15
3.3.- Fetch	15
3.3.1.- Petición get	16
3.3.2.- Petición delete	19
3.3.3.- Enviar datos al servidor post, put y patch	20
3.4.- Encadenamiento de promesas	23
3.4.- Refactorizando	25
4.- Async / Await.....	28

1.- Comunicación asíncrona en JavaScript

En el mundo del desarrollo web moderno, la **comunicación asíncrona** se ha convertido en una herramienta fundamental para crear aplicaciones fluidas, receptivas y escalables. A diferencia de **la comunicación síncrona, donde el programa se bloquea** hasta que una tarea se completa, **la comunicación asíncrona permite que el programa continúe ejecutando otras tareas mientras se espera a que una operación de larga duración finalice**. Esto se traduce en una mejor experiencia de usuario, ya que la interfaz no se congela mientras se cargan datos o se procesan peticiones.

La comunicación asíncrona significa que las tareas no se ejecutan en una secuencia estricta una tras otra, sino que pueden superponerse o ejecutarse simultáneamente. Esto es posible gracias al **modelo de ejecución de JavaScript**, que permite que el navegador ejecute varias tareas al mismo tiempo sin bloquear la interfaz de usuario.

Imaginad que estáis en una cafetería y queréis pedir un café y un pastel. En un modelo síncrono, tendríais que esperar a que el camarero os prepare el café antes de poder pedir el pastel. Sin embargo, en un modelo asíncrono, podríamos pedir el café al camarero y luego ir a la vitrina para elegir un pastel mientras el café se prepara. Una vez que el pastel esté listo, el camarero os lo dará y podréis disfrutar de ambos al mismo tiempo.

La comunicación asíncrona ofrece varias ventajas en el desarrollo web, entre las que podemos destacar:

- **Mejora la experiencia del usuario:** La interfaz de usuario no se congela mientras se esperan datos o se procesan peticiones, lo que se traduce en una experiencia más fluida y receptiva.
- **Aumenta la escalabilidad:** Las aplicaciones asíncronas pueden manejar un mayor número de peticiones simultáneas sin comprometer el rendimiento.

- **Simplifica el código:** La comunicación asíncrona puede hacer que el código sea más fácil de leer y mantener, ya que evita los bloqueos y permite que las tareas se ejecuten de forma independiente.

Por otro lado, la comunicación asíncrona presenta varios problemas:

- Gestionar múltiples peticiones asíncronas puede llevarnos muchos problemas, al ser asíncrono cada petición acabará en un tiempo indeterminado y seguro que no acaban en el orden en que se lanzan.
- Podemos tener peticiones asíncronas que dependan de la finalización de otra.
- Podemos querer lanzar un conjunto de peticiones asíncronas y querer esperar a que todas hayan acabado para poder realizar acciones con todos los datos recibidos.

La comunicación asíncrona se utiliza en una amplia variedad de aplicaciones web, como, por ejemplo:

- **Carga de datos:** Cuando un usuario carga una página web, se pueden cargar datos de forma asíncrona para evitar que la página se bloquee mientras se espera a que todos los datos estén disponibles.
- **Peticiones a APIs REST:** Las APIs REST se suelen consumir de forma asíncrona para obtener datos de servidores remotos.
- **Chat en tiempo real:** Las aplicaciones de chat en tiempo real utilizan la comunicación asíncrona para enviar y recibir mensajes de forma instantánea.
- **Juegos online:** Los juegos online necesitan una comunicación asíncrona para sincronizar las acciones de los jugadores en tiempo real.

JavaScript ofrece actualmente varios mecanismos para la comunicación asíncrona:

- Callbacks
- Promesas
- Async/await

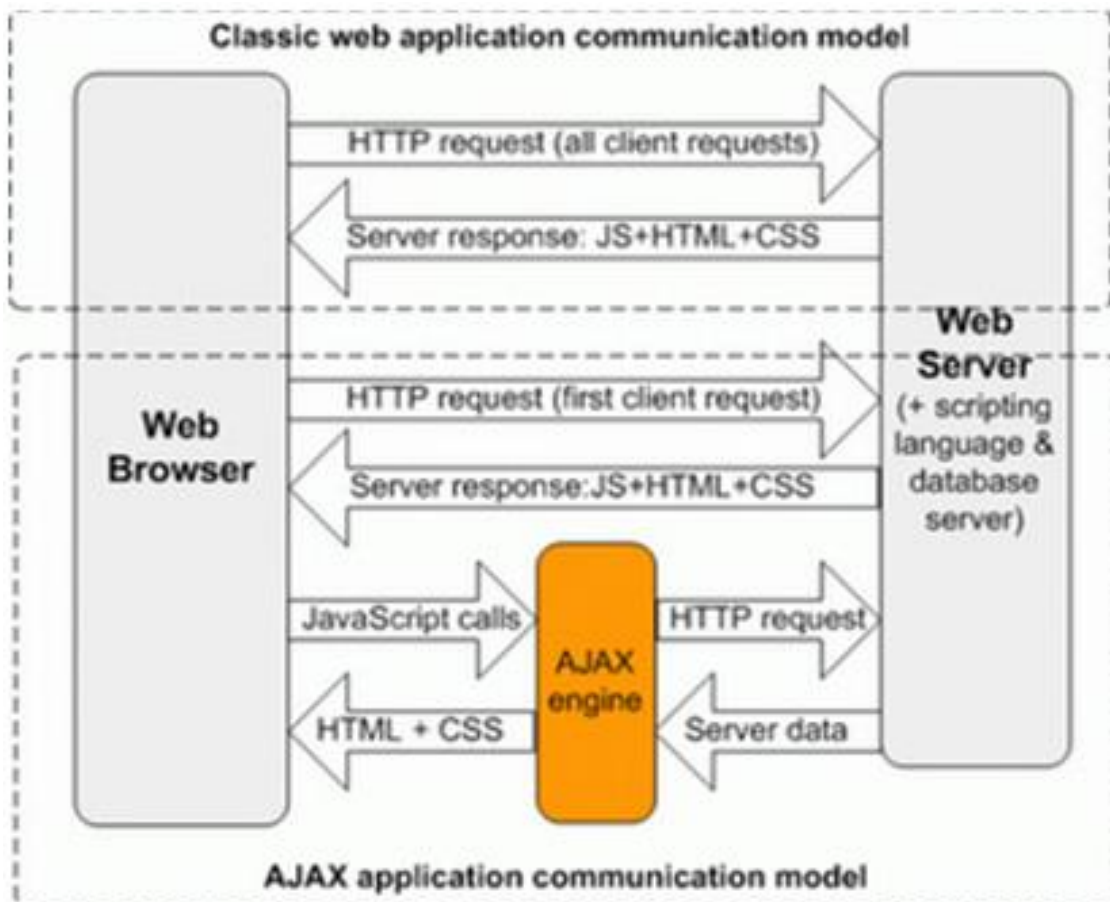
Independientemente del mecanismo que utilicemos por debajo de él esta AJAX.

2.- AJAX

JavaScript Asíncrono + XML (AJAX) no es una tecnología por sí misma, es un término que describe un nuevo modo de utilizar conjuntamente varias tecnologías existentes. Esto incluye: HTML o XHTML, CSS, JavaScript, DOM, XML, XSLT, y lo más importante, el objeto **XMLHttpRequest**.

Cuando estas tecnologías se combinan en un modelo AJAX, es posible lograr aplicaciones web capaces de actualizarse continuamente sin tener que volver a cargar la página completa. Esto crea aplicaciones más rápidas y con mejor respuesta a las acciones del usuario.

Comparación del modelo clásico de aplicaciones web frente al modelo utilizando ajax



XMLHttpRequest es un objeto de navegador incorporado que permite realizar solicitudes HTTP en JavaScript.

A pesar de tener la palabra "XML" en su nombre, puede trabajar con más formatos, no solo en formato XML. Podemos cargar/descargar archivos, comprobar el progreso de la petición y mucho más.

2.1- Funcionamiento del objeto XMLHttpRequest

El funcionamiento básico del objeto XMLHttpRequest es el siguiente:

1. Crear el objeto.
2. Inicializar el objeto
3. Enviar el objeto.
4. Esperar respuesta del servidor
5. Trabajar con la información recibida

Vamos a ver cada uno de los pasos en detalle:

1. Crear el objeto XMLHttpRequest:

```
let xhr = new XMLHttpRequest();
```

El constructor no tiene argumentos.

2. Inicializar el objeto

```
xhr.open(method, URL, [async, user, password])
```

Este método especifica los parámetros principales de la solicitud:

- **method** método HTTP. Por lo general "GET" o "POST".
- **URL** la URL de la petición
- **async** si se establece explícitamente en false, entonces la solicitud es síncrona
- **user, password** inicio de sesión y contraseña para la autenticación HTTP básica (si es necesario).

La llamada open, al contrario de su nombre, no abre la conexión. Solo configura la solicitud. La actividad de la red solo comienza con la llamada de **send**.

3. Enviar el objeto

```
xhr.send([body])
```

Este método abre la conexión y envía la solicitud al servidor. El parámetro `body` es opcional contiene el cuerpo de la solicitud.

Algunos métodos de solicitud como GET o DELETE no tienen cuerpo. Otros como POST, PUT o PATCH **necesitan usar body** para enviar los datos al servidor.

4. Esperar respuesta del servidor

Una vez enviada la petición el objeto XMLHttpRequest estará “**escuchando**” lo que va sucediendo con el estado de la petición, para ello utiliza los **eventos** del objeto xhr.

Los eventos más utilizados son:

- **load** cuando se completa la solicitud (incluso si el estado HTTP es como 400 o 500), y la respuesta se descarga completamente.

```
xhr.onload = function() {  
    alert(`Loaded: ${xhr.status} ${xhr.response}`);  
};
```

- **error** cuando no se pudo realizar la solicitud, por ejemplo, red inactiva o URL no válida.

```
xhr.onerror = function() {  
    alert(`Network Error`);  
};
```

5. Trabajar con la información recibida

Una vez que el servidor ha respondido, podemos recibir el resultado en las siguientes propiedades del objeto xhr:

- **status**
HTTP código de estado (un número): 200, 201, 404, 403 y así sucesivamente.
- **statusText**
Mensaje de estado HTTP (una cadena): generalmente OK para 200, Not Found para 404, Forbidden para, 403 etc.
- **response**
La respuesta del servidor, aquí es donde estará la información que nos manda el servidor. Existen diferentes formatos de respuesta del

servidor. Podemos usar la propiedad **xhr.responseType** para establecer el formato de respuesta:

- "" (predeterminado): obtener como cadena
- "text" llega como string
- "arraybuffer" llega como ArrayBuffer para datos binarios
- "blob" usado para datos binarios
- "document" llega como documento XML (puede usar XPath y otros métodos XML),
- "json" llega como JSON (analizado automáticamente).

Hoy en día lo más habitual es trabajar con API REST que nos devolverán los datos en un formato JSON.

Para poder especificar que nuestro objeto xhr recibirá datos en JSON se lo deberemos indicar.

```
xhr.responseType = 'json';
```

Con lo cual una petición completa quedaría así:

```
let xhr = new XMLHttpRequest();
xhr.open(method, URL, [async, user, password])
xhr.responseType = 'json';
xhr.send([body])

xhr.onload = function() {
    alert(`Loaded: ${xhr.status} ${xhr.response}`);
};

//Error en petición
xhr.onerror = function() {
    alert(`Network Error`);
};
```

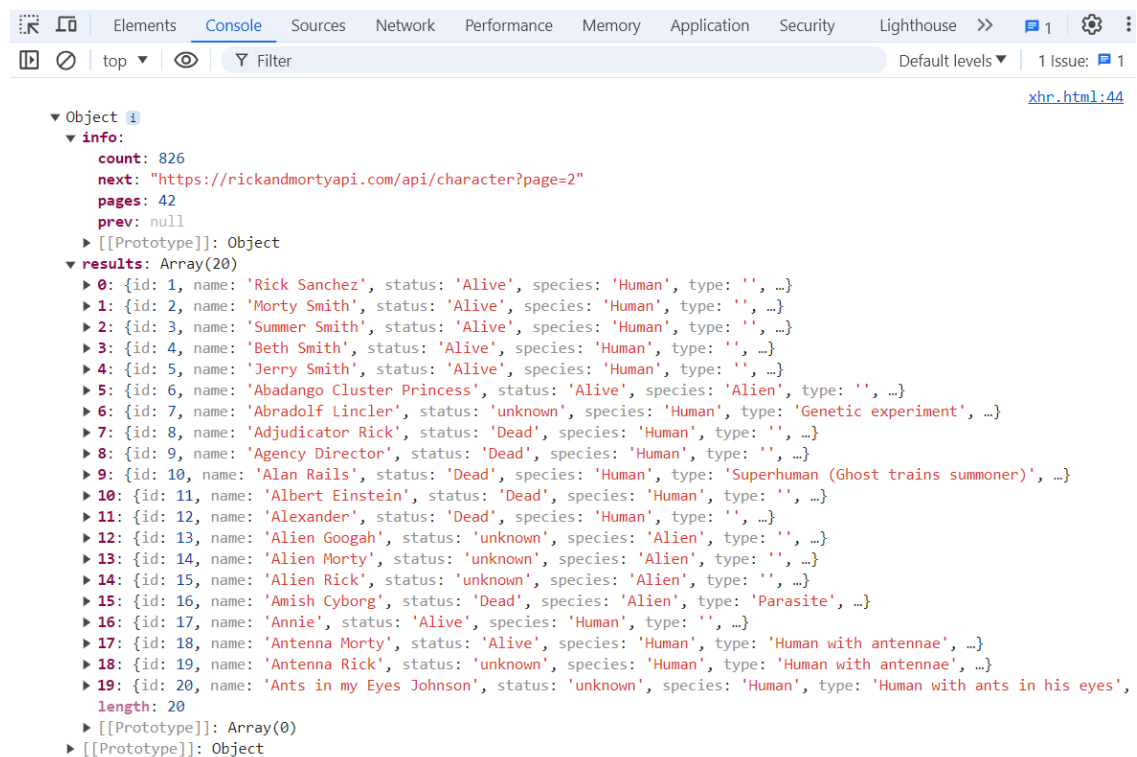
2.2- Petición get

Una vez visto el funcionamiento básico del objeto XMLHttpRequest vamos a realizar una petición get completa, para ello necesitaríamos acceder a una API, usaremos la API de **Rick and Morty** que nos ofrece información sobre esta serie. La web oficial es <https://rickandmortyapi.com>

Si vamos a **Doc nos informa** de las opciones de la API. Nosotros vamos a hacer una petición para obtener todos los personajes de la serie, para ello deberemos acceder al endpoint <https://rickandmortyapi.com/api/character>

Disponemos en la información del esquema devuelto por la petición get para el endpoint character, es muy completo, podemos comprobarlo en el navegador introduciendo la url.

Si hacemos bien la petición obtendremos el mismo resultado que en el navegador



```
▼ Object 1
  ▼ info:
    count: 826
    next: "https://rickandmortyapi.com/api/character?page=2"
    pages: 42
    prev: null
    ► [[Prototype]]: Object
  ▼ results: Array(20)
    ► {id: 1, name: 'Rick Sanchez', status: 'Alive', species: 'Human', type: '', ...}
    ► {id: 2, name: 'Morty Smith', status: 'Alive', species: 'Human', type: '', ...}
    ► {id: 3, name: 'Summer Smith', status: 'Alive', species: 'Human', type: '', ...}
    ► {id: 4, name: 'Beth Smith', status: 'Alive', species: 'Human', type: '', ...}
    ► {id: 5, name: 'Jerry Smith', status: 'Alive', species: 'Human', type: '', ...}
    ► {id: 6, name: 'Abadango Cluster Princess', status: 'Alive', species: 'Alien', type: '', ...}
    ► {id: 7, name: 'Abradolf Lincler', status: 'unknown', species: 'Human', type: 'Genetic experiment', ...}
    ► {id: 8, name: 'Adjudicator Rick', status: 'Dead', species: 'Human', type: '', ...}
    ► {id: 9, name: 'Agency Director', status: 'Dead', species: 'Human', type: '', ...}
    ► {id: 10, name: 'Alan Rails', status: 'Dead', species: 'Human', type: 'Superhuman (Ghost trains summoner)', ...}
    ► {id: 11, name: 'Albert Einstein', status: 'Dead', species: 'Human', type: '', ...}
    ► {id: 12, name: 'Alexander', status: 'Dead', species: 'Human', type: '', ...}
    ► {id: 13, name: 'Alien Googah', status: 'unknown', species: 'Alien', type: '', ...}
    ► {id: 14, name: 'Alien Morty', status: 'unknown', species: 'Alien', type: '', ...}
    ► {id: 15, name: 'Alien Rick', status: 'unknown', species: 'Alien', type: '', ...}
    ► {id: 16, name: 'Amish Cyborg', status: 'Dead', species: 'Alien', type: 'Parasite', ...}
    ► {id: 17, name: 'Annie', status: 'Alive', species: 'Human', type: '', ...}
    ► {id: 18, name: 'Antenna Morty', status: 'Alive', species: 'Human', type: 'Human with antennae', ...}
    ► {id: 19, name: 'Antenna Rick', status: 'unknown', species: 'Human', type: 'Human with antennae', ...}
    ► {id: 20, name: 'Ants in my Eyes Johnson', status: 'unknown', species: 'Human', type: 'Human with ants in his eyes', ...}
    length: 20
    ► [[Prototype]]: Array(0)
  ► [[Prototype]]: Object
```

La ventaja es que nosotros ahora con esos datos podemos hacer lo que queramos.

Vamos a hacer una interface sencilla que nos muestre los nombres de los personajes en una lista, para ello en nuestra web añadiremos un botón para poder consultar los personajes

Personajes de Rick and Morty

[Ver personajes](#)

- Rick Sanchez
- Morty Smith
- Summer Smith
- Beth Smith
- Jerry Smith
- Abadango Cluster Princess
- Abradolf Lincler
- Adjudicator Rick
- Agency Director
- Alan Rails
- Albert Einstein
- Alexander
- Alien Googah
- Alien Morty
- Alien Rick
- Amish Cyborg
- Annie
- Antenna Morty
- Antenna Rick
- Ants in my Eyes Johnson

Si hubiéramos querido hacer una petición a un character deberemos indicarlo en la url, por ejemplo, para obtener el character 5 deberemos hacer una petición get a la url <https://rickandmortyapi.com/api/character/5>

Vamos a hacer una interface que al pulsar sobre un botón nos pida por un id de personaje y nos muestre sus datos en un card.

La página de inicio nos muestra los dos botones

Personajes de Rick and Morty

[Ver personajes](#)[Ver un personaje](#)

Seleccionamos el personaje con su id en un promp

Esta página dice

Dime el id del personaje...

[Aceptar](#) [Cancelar](#)

Realizamos la petición al servidor y mostramos al personaje en el card

Personajes de Rick and Morty

[Ver personajes](#)[Ver un personaje](#)

Cowboy Rick

2.3- Petición delete

En este caso, la petición delete tiene la misma estructura que una petición get para consultar un registro, pero utilizando el método delete. Si el registro a borrar no existe la petición devolverá un código HTTP de error.

En nuestra API REST para **añadir eliminar registro** en la entidad artículos:

```
const url="http://localhost:3000/articulos"

function deleteArticulo(){
  let id=prompt("Dime el id del articulo...")
  let xhr = new XMLHttpRequest();
  xhr.open("delete", url + "/" + id)
  xhr.responseType = 'json';
  xhr.send()

  xhr.onload = function() {
    if (xhr.status==200){
      console.log("El articulo se ha borrado")
    }
    else{
      console.log("ERROR. El articulo NO se ha borrado")
    }
  };

  xhr.onerror = function() {
    alert(`Network Error`);
  };
}
```

2.4- Enviar datos al servidor post, put y patch

La petición get junto al delete son las más sencillas pues no mandamos información nueva al servidor por eso en el método send no incluimos nada, lo dejamos vacío. En cambio, en las peticiones post, put y patch sí que mandamos datos al servidor, en el post el nuevo registro y el put y el patch los valores del registro a modificar. Es por ello que ahora sí que deberemos **mandar el registro en el método send** de la petición.

Además, como estamos mandando información al servidor deberemos decirle que tipo de información le estamos mandando. Para ello deberemos de utilizar una cabecera HTTP que se lo indique, utilizaremos **setRequestHeader(name, value)** que establece el encabezado de solicitud con el dado name y value.

En nuestro caso para mandar un registro en formato json y utf-8 será:

```
setRequestHeader('Content-type', 'application/json; charset=utf-8');
```

Para trabajar con una API REST lo normal es que el registro que mandemos sea un objeto JSON, que construiremos con los valores de cada campo, luego lo que haremos es **mandar el objeto en un formato JSON**, para ello deberemos:

1. Establecer el encabezado **Content-Type: application/json**
2. Convertir el objeto a cadena de texto para poder mandarlo por la red, para ello utilizaremos **JSON.stringify(objeto)**.

En nuestra API REST para **añadir un nuevo registro** en la entidad artículos:

```
const url="http://localhost:3000/articulos"

function postArticulo(){
  let id=prompt("Dime el id del articulo...")
  let nombre=prompt("Dime el nombre del articulo...")
  let precio=parseInt(prompt("Dime el precio del articulo..."))

  let articulo=JSON.stringify({"id":id,"nombre":nombre,"precio":precio})

  let xhr = new XMLHttpRequest();
  xhr.open("post", url)

  xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');
  xhr.send(articulo)
```

```
xhr.onload = function() {  
  if (xhr.status==201){  
    console.log("El articulo se ha insertado.")  
  }  
  else{  
    console.log("ERROR. El articulo NO se ha insertado.")  
  };  
  xhr.onerror = function() {  
    alert(`Network Error`);  
  };  
};  
}
```

En nuestra API REST para **añadir modificar un registro** en la entidad artículos:

```
const url="http://localhost:3000/articulos"  
  
function putArticulo(){  
  let id=prompt("Dime el id del articulo a modificar...")  
  let nombre=prompt("Dime el nuevo nombre del articulo...")  
  let precio=parseInt(prompt("Dime el nuevo precio del articulo..."))  
  
  let articulo=JSON.stringify({"id":id,"nombre":nombre,"precio":precio})  
  
  let xhr = new XMLHttpRequest();  
  xhr.open("put", url + "/" + id)  
  xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');  
  xhr.send(articulo)  
  
  xhr.onload = function() {  
    if (xhr.status==200){  
      console.log("El articulo ha sido modificado.")  
    }  
    else{  
      console.log("ERROR. El articulo NO se ha modificado.")  
    };  
  };  
  
  xhr.onerror = function() {  
    alert(`Network Error`);  
  }  
}
```

3.- Promesas

La forma de trabajar que hemos visto con el objeto **XMLHttpRequest** trabaja **con callbacks**, es decir que se pasa una función para gestionar la petición. Para peticiones sencillas es fácil de gestionar, pero **el problema** viene cuando se tienen que implementar **peticiones anidadas**, es decir cuando tenemos varias peticiones que ejecutar, pero para que cada una se ejecute debe de esperar a la anterior porque necesita el dato que le devuelve.

Por ejemplo, en nuestra API REST podía ser una petición que para un proveedor nos devolviera el precio del artículo que vende. Para realizar esto deberemos hacer:

- una **petición get a la entidad proveedores** para obtener el proveedor,
- una vez que tenemos el proveedor deberemos sacar el id del artículo que vende y con ese id hacer una **petición get a la entidad artículos**.

Para resolver esto tenemos que hacer dos peticiones get, pero la segunda petición ha de esperar a que finalice la primera. Como son asíncronas no podemos lanzarlas a la vez, para resolver este problema **en el onload de la primera petición get es donde deberíamos de realizar la segunda petición**.

Si en vez de dos peticiones tuviéramos más el código sería muy difícil de interpretar y mantener, llegando al caso conocido como **callback Hell or Pyramid of Doom**

```
1 // Callback Hell
2
3
4 a(function (resultsFromA) {
5     b(resultsFromA, function (resultsFromB) {
6         c(resultsFromB, function (resultsFromC) {
7             d(resultsFromC, function (resultsFromD) {
8                 e(resultsFromD, function (resultsFromE) {
9                     f(resultsFromE, function (resultsFromF) {
10                         console.log(resultsFromF);
11                     })
12                 })
13             })
14         })
15     })
16 });
```

Las promesas se incorporaron de forma oficial al lenguaje JavaScript en ECMAScript 6 (ES6). Esta versión del estándar introdujo muchas características nuevas y modernas que cambiaron significativamente la forma en que escribimos JavaScript, y las promesas fueron una de las más importantes.

Una promesa es un objeto que representa **la terminación o el fracaso** de una **operación asíncrona**. Dado que la mayoría de los programadores **consumen promesas ya creadas**, es importante conocer como consumirlas tanto o más que incluso saber cómo crearlas.

Las características de una promesa son:

- **Asíncrona:** Significa que no bloquea la ejecución del código.
- **Finalización:** La promesa puede **cumplirse (resolve)** o **rechazarse (reject)**. Si se cumple, significa que la operación se realizó correctamente y tenemos el resultado. Si se rechaza, significa que ocurrió un error.

¿Por qué son útiles las promesas?

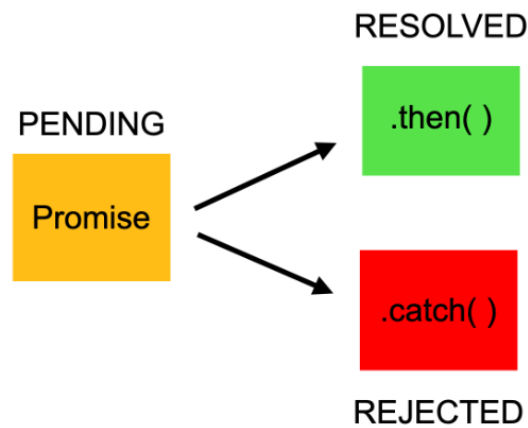
- **Manejo de código asíncrono:** Facilitan la gestión de operaciones que tardan en completarse, como peticiones a servidores, operaciones de entrada/salida, etc.
- **Encadenamiento de operaciones:** Permiten encadenar múltiples operaciones asíncronas de manera secuencial.
- **Mejor legibilidad:** Hacen el código más limpio y fácil de entender en comparación con las tradicionales callbacks.

Nosotros **vamos a ser consumidores de promesas**, es decir vamos a utilizar elementos que están basados en promesas, con lo cual no vamos a ver cómo construir una promesa.

Una Promesa es un objeto. Hay 3 estados del objeto Promesa:

- **Pendiente:** Estado inicial, antes de que la promesa tenga éxito o falle
- **Resuelto:** Promesa completada
- **Rechazado:** Promesa fallida

Al lanzar la promesa el estado será Pendiente, más tarde la promesa finalizará, si acaba de forma correcta el estado será **Resolved** y lo gestionará **.then()**, si por el contrario la promesa finaliza con error el estado será **Rejected** y lo gestionará **.catch()**



Las funciones de consumo pueden registrarse (suscribirse) utilizando los métodos **.then**, **.catch**

3.1.- Then

Es el método que nos permite gestionar la finalización de una forma correcta de la promesa.

La sintaxis es:

```
promise.then( function(result) { /* gestión resultados */ })
```

Lo más habitual es usar dentro del then la sintaxis fatArrow

```
promise.then( result => { /* gestión resultados */ })
```

En este caso, al acabar de forma correcta la promesa devolverá un resultado, para gestionar ese resultado utilizamos el método then que lo que tiene como parámetro es una función que recibe el resultado de la promesa para poder gestionarlo.

3.2.- Catch

Es el método que nos permite gestionar la finalización con error de la promesa.

La sintaxis es:

```
promise.catch( function(error) { /* gestión error */ })
```

Lo más habitual es usar dentro del catch la sintaxis fatArrow

```
promise.catch( error => { /* gestión error */ })
```

En este caso, al acabar con error, la promesa devolverá un error, para gestionarlo utilizamos el método catch que lo que tiene como parámetro es una función que recibe el error de la promesa para poder gestionarlo.

3.3.- Fetch

Fetch es una API de JavaScript que nos permite realizar peticiones HTTP de forma asíncrona. Es el estándar para peticiones web en JavaScript que reemplaza al objeto XMLHttpRequest.

¿Por qué es tan útil Fetch?

- **Sencillez:** Proporciona una interfaz sencilla para hacer peticiones, haciendo que el código sea más legible y fácil de mantener.
- **Promesas:** Utiliza promesas, lo que significa que puedes manejar el éxito o el fracaso de una petición de forma elegante y clara.
- **Flexibilidad:** Permite personalizar las peticiones con diferentes métodos HTTP (GET, POST, PUT, DELETE, etc.), encabezados y cuerpos.

Con Fetch ya no utilizaremos directamente el objeto XMLHttpRequest ya que fetch es una API que gestiona la petición HTTP mediante promesas, fetch se encarga de crear, configurar y mandar la petición HTTP y nos devolverá el resultado de la petición que gestionaremos con los métodos then y catch.

La función **fetch()** acepta dos parámetros:

1. La URL a la que enviar la petición (este es un parámetro obligatorio).
2. Las opciones a configurar en la petición. Podemos configurar el método de solicitud aquí (este es un parámetro opcional).

La función `fetch()` devuelve una Promesa:

- cuando la petición devuelve una respuesta, se llamará al método `then()`
- si la solicitud devuelve un error, se ejecutará el método `catch()`

```
fetch('url', {})  
  .then(response => {  
    // Gestionamos la respuesta de la petición aquí  
  })  
  .catch(error => {  
    // Si hay un error de red lo gestionamos aquí  
  })
```

El método `.catch()` se puede omitir en Fetch API. Se usa solo cuando Fetch no puede realizar una solicitud a la API, como por ejemplo si no hay conexión de red o no se encuentra la URL.

3.3.1.- Petición get

Para realizar una petición get únicamente debemos indicar la url a la que queremos acceder y gestionar el resultado de la petición.

```
let url='http://localhost:3000/articulos'  
  
fetch(url, {})  
  .then(response => console.log(respuesta))  
  .catch(error => console.log("Network error"))
```

El resultado es:

```
▼ Response {type: 'cors', url: 'http://localhost:3000/articulos',  
  redirected: false, status: 200, ok: true, ...} ⓘ  
  ► body: ReadableStream  
    bodyUsed: false  
  ► headers: Headers {}  
    ok: true  
    redirected: false  
    status: 200  
    statusText: "OK"  
    type: "cors"  
    url: "http://localhost:3000/articulos"  
  ► [[Prototype]]: Response
```

La promesa nos devuelve un objeto **Response**, que encapsula toda la información sobre la respuesta del servidor, incluyendo:

- **status:** Un número entero que indica el código de estado HTTP de la respuesta. Por ejemplo, 200 para éxito, 404 para no encontrado, 500 para error del servidor, etc.
- **statusText:** Una cadena de texto que describe el estado HTTP en palabras (por ejemplo, "OK", "Not Found", "Internal Server Error").
- **headers:** Un objeto que contiene los encabezados de la respuesta. Estos encabezados proporcionan información adicional sobre la respuesta, como el tipo de contenido, la codificación, etc.
- **body:** El cuerpo de la respuesta. Puede ser texto, JSON, un blob, o cualquier otro tipo de datos. Para acceder al cuerpo, debes utilizar métodos como `text()`, `json()`, `blob()`, etc.
- **ok:** Un booleano que indica si la petición fue exitosa (código de estado entre 200 y 299).

Podemos ver que la **propiedad body contiene un ReadableStream**. Para usar `ReadableStream` en nuestra aplicación JavaScript, necesitamos convertirlo en un objeto json, para ello utilizaremos el método **`json()`**.

El método **`json()`** se encarga de convertir la respuesta del servidor, que está en formato de texto, a un objeto JavaScript. **Este proceso también es asíncrono**, ya que implica un análisis del texto y la construcción del objeto, con lo cual podemos encadenar otro `then`.

```
let url='http://localhost:3000/articulos'

fetch(url , {})
  .then(response => response.json())
  .then(articulos => console.log(articulos))
  .catch(error    => console.log("Network error"))
```

El resultado ahora será:

```
▼ (3) [{...}, {...}, {...}] ⓘ
  ▶ 0: {id: '1', nombre: 'Portátil Acer Aspire 5', precio: 700}
  ▶ 1: {id: '2', nombre: 'Monitor LG 27 pulgadas', precio: 300}
  ▶ 2: {id: '3', nombre: 'Teclado mecánico Corsair K65', precio: 150}
    length: 3
  ▶ [[Prototype]]: Array(0)

>
```

Si hacemos la petición para un artículo en concreto, por ejemplo, el 2

```
let url='http://localhost:3000/articulos/2'

fetch(url , {})
  .then(response => response.json())
  .then(articulo => console.log(articulo))
  .catch(error   => console.log("Network error"))
```

Hasta ahora hemos realizado dos peticiones sencillas y que sabemos que no dan error. **Fetch gestiona el resultado de la petición, tanto si es correcta o como si da error en el método then y deja el catch para errores de red**

Si queremos comprobar los errores, lo más habitual es:

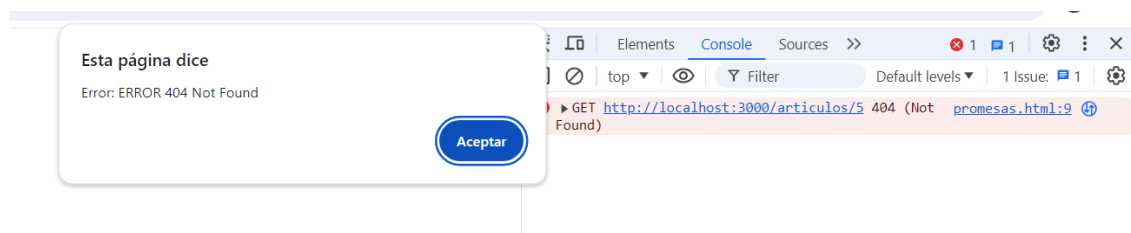
- comprobar dentro del then que la petición ha ido bien, para ello utilizaremos la propiedad ok.
- si ha habido error podremos acceder al error mediante las propiedades status y statusText y generamos un error con esos datos. Ese error será recogido por el método catch
- si ha ido bien la petición devolveremos los datos en formato json

Vamos a hacer una petición al artículo con id 5, que no existe y debería de darnos un error

```
let url='http://localhost:3000/articulos/5'

fetch(url,{})
  .then(response => {
    if (!response.ok) {
      throw new Error(`Mi Error ${response.status} ${response.statusText}`);
    }
    return response.json();
  })
  .then(articulo => console.log(articulo))
  .catch(error => alert(error))
```

Ahora al intentar acceder al registro con id 5



3.3.2.- Petición delete

Al igual que sucedía con el objeto xhr la petición delete es similar a la petición get. Accederemos a la entidad a eliminar indicando su id, si el registro no existe se producirá un error 404, si el registro a borrar existe, se eliminará.

Con el objeto xhr indicábamos el tipo de petición en el método open, con fetch para indicar un método distinto al get, que es el predeterminado, deberemos utilizar el segundo parámetro que es opcional.

Este parámetro es un objeto que indica las opciones de la petición, principalmente:

- El tipo de petición (no necesario para get)
- Las cabeceras de la petición
- El body de la petición (para peticiones post, put y patch)

En nuestro caso sólo deberemos de indicar el método, ya que no vamos a mandar información al servidor.

Si quisiéramos borrar el artículo con id 2

```
let url='http://localhost:3000/articulos/2'

fetch(url,{method: 'DELETE'})
.then(response => {
  if (!response.ok) {
    throw new Error(`Mi Error ${response.status} ${response.statusText}`);
  }
  return response.json();
})
.then(articulo => console.log(articulo) )
.catch(error => alert(error))
```

En este caso se ha borrado el artículo con id 2, json-server nos devolverá un objeto vacío.

3.3.3- Enviar datos al servidor post, put y patch

Las peticiones post, put y patch **mandan al servidor un objeto**, ya sea para insertar uno nuevo o para modificar un registro existente. Además, como estamos mandando información al servidor deberemos decirle que tipo de información le estamos mandando, para ello deberemos de **utilizar una cabecera HTTP** que se lo indique.

Para trabajar con una API REST lo normal es que el registro que mandemos sea un objeto JSON, que construiremos con los valores de cada campo, luego lo que haremos es **mandar el objeto en un formato JSON**.

Todas estas acciones las deberemos indicar en el segundo parámetro de la petición fetch, en las opciones de la petición.

Para una petición post

1. Indicar la url de la entidad
2. Indicar el método Post
3. Establecer el encabezado **Content-Type: application/json**
4. Obtener un registro nuevo y convertir el objeto a cadena de texto para poder mandarlo por la red en el body de la petición, para ello utilizaremos **JSON.stringify(objeto)**.

```
let url='http://localhost:3000/articulos'

let id=prompt("Dime el id: ")
let nombre=prompt("Dime el nombre: ")
let precio=parseInt(prompt("Dime el precio: "))

let
articuloNuevo=JSON.stringify({"id":id,"nombre":nombre,"precio":precio})

let opciones={method: 'POST',
              headers: {'Content-Type': 'application/json'},
              body: articuloNuevo
            }
fetch(url,opciones)
  .then(response => {
    if (!response.ok) {
      throw new Error(`Mi Error ${response.status} ${response.statusText}`);
    }
    return response.json();
  })
  .then (articulo => console.log(articulo))
  .catch(error => alert(error))
```

En este caso insertaremos un registro nuevo con los datos indicados en el objeto articuloNuevo. La respuesta puede ser:

- **si el objeto no existía**, se insertará y json-server devolverá el objeto insertado.
- **si el objeto ya existía**, se devolverá un error 500 indicando que hay un duplicado con ese id

Para una petición put

1. Indicar la url de la entidad y el id del objeto que queremos modificar los datos
2. Indicar el método Put
3. Establecer el encabezado **Content-Type: application/json**
4. Obtener un registro con los datos a modificar y convertir el objeto a cadena de texto para poder mandarlo por la red en el body de la petición, para ello utilizaremos **JSON.stringify(objeto)**.

```
let url='http://localhost:3000/articulos'

let id=prompt("Dime el id del registro a modificar: ")
let nombre=prompt("Dime el nuevo nombre: ")
let precio=parseInt(prompt("Dime el nuevo precio: "))

let articuloMod=JSON.stringify({"id":id,"nombre":nombre,"precio":precio})

let opciones={method: 'PUT',
               headers: {'Content-Type': 'application/json'},
               body: articuloMod
            }
fetch(url + "/" + id, opciones)
  .then(response => {
    if (!response.ok) {
      throw new Error(`Mi Error ${response.status} ${response.statusText}`);
    }
    return response.json();
  })
  .then (articulo => console.log(articulo))
  .catch(error => alert(error))
```

En este caso modificaremos los datos del objeto con el id indicado, con los datos indicados en el objeto articuloMod.

La respuesta puede ser:

- **si el objeto existía**, se modificará y json-server devolverá el objeto modificado.
- **si el objeto no existía**, se devolverá un error 404 indicando que no existe un objeto con ese id

3.4.- Encadenamiento de promesas

Hasta ahora estamos trabajando con peticiones individuales, es decir, **estamos realizando peticiones que no dependen de ninguna otra**. Si quisiéramos acceder a los **datos del artículo que vende un proveedor** tendríamos que **realizar dos peticiones get**, una para acceder a los datos del proveedor y una vez que tuviéramos los datos del proveedor acceder con el `idArticulo` a los datos del artículo. Además, **la segunda petición debe de realizarse cuando haya finalizado la primera**. Este problema es muy común en peticiones asíncronas y como ya vimos con el objeto `xhr` y las `callback` podemos llegar al **callback Hell**. Veamos una solución con promesas.

```
let url='http://localhost:3000/'
let idProveedor=prompt("Dime el id del proveedor: ")

fetch(url + "proveedores" + "/" + idProveedor)
  .then(response => {
    if (!response.ok) {
      throw new Error(`Error proveedor ${response.status}
                      ${response.statusText}`);
    }
    return response.json();
  })
  .then(p => {console.log(p)
    fetch(url + "articulos" + "/" + p.idArticulo)
      .then(response => {
        if (!response.ok) {
          throw new Error(`Error articulo ${response.status}
                          ${response.statusText}`);
        }
        return response.json();
      })
      .then(a=>console.log(a))
      .catch(error=>alert(error))
  })
  .catch(error => alert(error))
```

En resumen, lo que hacemos es:

- Realizamos una petición `get` a la entidad `proveedores`, gestionamos la respuesta con `then` y el error con `catch`

- En el then de la petición satisfactoria a proveedores hacemos otra petición get a artículos con el idArticulos que sacamos del proveedor, gestionamos la petición con then y con catch.

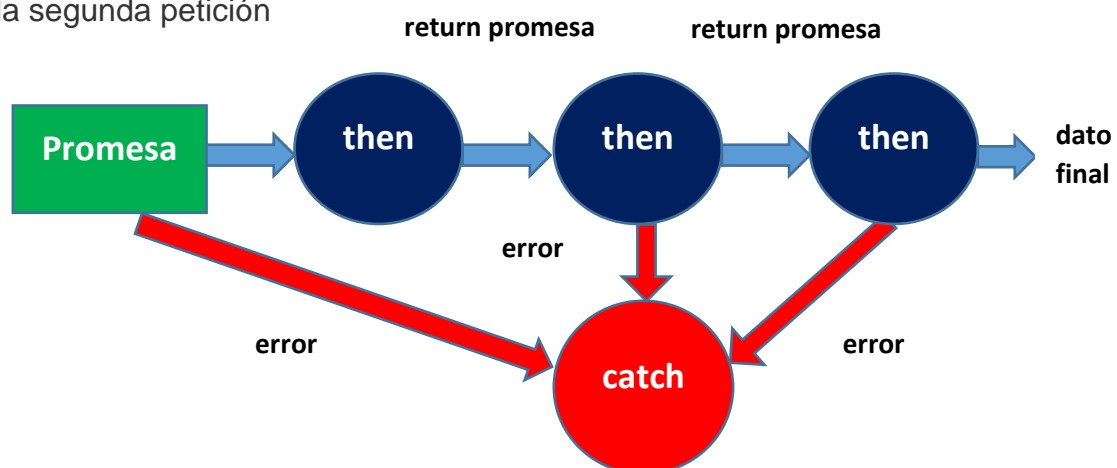
Aunque el código es más claro que con el objeto xhr y con las callback, la estructura del código es similar, es decir, estamos realizando una petición dentro de otra y duplicando los then y los catch por cada petición. Podríamos decir que estamos realizando promesas anidadas, una dentro de otra

Una de las ventajas del uso de promesas es el encadenamiento de promesas. El encadenamiento de promesas implica utilizar los métodos then() y catch() para manejar el resultado o el error de una promesa y devolver otra promesa. Esta técnica permite ejecutar acciones secuenciales de manera clara y legible.

La esencia del encadenamiento es que cada then devuelva una promesa de esta manera podemos ir pasando los resultados de una promesa al siguiente then y además solo utilizaremos un catch.

Esto ya lo estábamos utilizando hasta ahora, cada vez que hacemos una petición fetch los datos que recibimos en el response los pasamos a otro then para poder obtener la información mediante el método .json.

La idea ahora será la misma, los datos obtenidos en la primera petición no los gestionaremos en su then, sino que devolveremos el resultado de la primera promesa para que sea gestionado por el siguiente fetch, que será el que realice la segunda petición



Utilizando el encadenamiento quedaría de la siguiente manera.

```
let url='http://localhost:3000/'
let idProveedor=prompt("Dime el id del proveedor: ")

fetch(url + "proveedores" + "/" + idProveedor)
  .then(response => {
    if (!response.ok) {
      throw new Error(`Error proveedor ${response.status}
                      ${response.statusText}`);
    }
    return response.json();
  })
  .then(p => {console.log(p)
    return fetch(url + "articulos" + "/" + p.idArticulo)
      .then(response => {
        if (!response.ok) {
          throw new Error(`Error articulo
                          ${response.status} ${response.statusText}`);
        }
        return response.json();
      })
  })
  .then(a=>console.log(a))
  .catch(error => alert(error))
```

De esta manera es mucho más sencillo leer el código, cada petición devuelve los datos al siguiente then y así sucesivamente por cada petición, en el momento que haya un error se va directamente al catch, únicamente habrá un manejador catch que gestionara todos los errores que se produzcan independientemente de en qué petición suceda.

3.4.- Refactorizando

Hasta ahora hemos visto cómo funcionan las promesas, como podemos hacer una petición de forma asíncrona al servidor mediante fetch y cómo podemos encadenar varias peticiones. En una aplicación real es casi seguro que deseemos hacer varias peticiones a la misma entidad, hasta ahora estábamos escribiendo el código para una petición en concreto, con lo cual si quisiéramos volver a hacer otra petición igual tendríamos que volver a escribir todo el código. Es lo mismo que nos sucede cuando empezamos a programar y queremos utilizar el mismo código varias veces, para ello utilizamos las funciones.

En nuestro caso lo que vamos a hacer es encapsular las peticiones fetch en una función que devuelva la petición, de esta manera podremos utilizar la petición todas las veces que queramos.

Vamos a diseñar una función para realizar peticiones get a la entidad articulo

```
function getArticulo(id){
  return fetch(url+"articulos"+"/"+id)
    .then(response => {
      if (!response.ok) {
        throw new Error(`Error en articulo ${response.status}
                        ${response.statusText}`);
      }
      return response.json()
    });
}
```

Hemos implementado una función llamada getArticulo que acepta como parámetro un id y que si la petición va bien devuelve los datos en formato json y que si va mal devolverá un error. Como la propia función devuelve directamente una promesa (sólo tiene una instrucción que es devolver la promesa generada con fetch) podremos llamarla y gestionarla como una promesa, es decir podremos usar then y catch. La llamada sería así

```
getArticulo(2)
  .then (a=>console.log(a))
  .catch(error => alert(error))
```

De esta forma estaríamos haciendo una petición al artículo con id 2.

Podríamos hacer lo mismo para proveedores

```
function getProveedor(id){
  return fetch(url+"proveedores"+"/"+id)
    .then(response => {
      if (!response.ok) {
        throw new Error(`error en Proveedor ${response.status}
                        ${response.statusText}`);
      }
      return response.json()
    });
}
```

La llamada quedaría

```
getProveedor(3)
  .then (a=>console.log(a))
  .catch(error => alert(error))
```

También podríamos realizar la petición anidada para poder obtener los datos del artículo que vende un proveedor

```
getProveedor(1)
  .then (p=>{console.log(p)
              return getArticulo(p.idArticulo)
            })
  .then(a=>console.log(a))
  .catch(error => alert(error))
```

Queda claro que se simplifica muchísimo el código y la legibilidad del mismo.

Con la refactorización podríamos diseñar nuestras propias funciones pasando los parámetros que necesitaríamos para implementar las utilidades de nuestra aplicación. Por ejemplo, aun podríamos refactorizar mas nuestro ejemplo y diseñar una función única para poder acceder a cualquier entidad y a cualquier objeto dentro de ella. En este caso deberíamos de implementar una función llamada por ejemplo getEntidad que aceptara dos parámetros, la entidad y el id.

```
function getEntidad(entidad,id){
  return fetch(url + entidad + "/" + id)
    .then(response => {
      if (!response.ok) {
        throw new Error(`error en ${entidad} con id
                        ${id} ${response.status}
                        ${response.statusText}`);
      }
      return response.json()
    });
}
```

4.- Async / Await

En el ES8 se introdujo como novedad las palabras clave **async/await**, que no son más que una forma de **azúcar sintáctico** para gestionar las promesas de una forma más similar a lo que solemos estar acostumbrados.

Con async/await seguimos manejando promesas, sin embargo, hay ciertos cambios importantes:

- No encadenamos mediante `.then()`, sino que usamos un sistema más tradicional.
- No disponemos del manejador `.catch()`
- Abandonamos el modelo **no bloqueante** y pasamos a uno **bloqueante**.

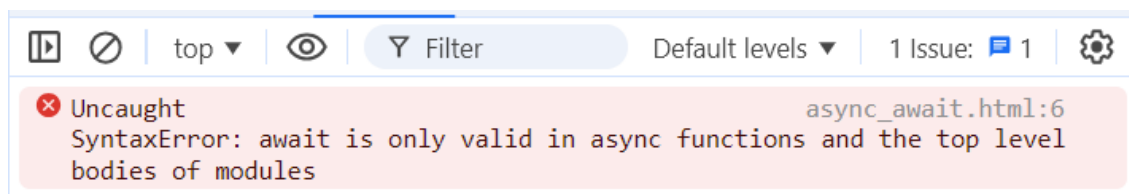
El funcionamiento de async/await está basado en el bloqueo de la petición, es decir, hasta que no se resuelva la petición no se pasa a la siguiente instrucción, de esta manera detenemos la ejecución del código hasta que se ha resuelto la petición.

Para obtener los artículos se podría hacer de la siguiente forma

```
let url='http://localhost:3000/articulos'  
  
const response=await fetch(url)  
const articulos= await response.json();  
console.log(articulos)
```

Ya no usamos el manejador `.then()`, en su lugar utilizamos una variable en la que guardamos el resultado de la petición `fetch`, para evitar que se pase a la siguiente instrucción antes de que se haya resuelto la promesa utilizamos la palabra `await`.

Este código nos devolverá el siguiente error



Este error lo que nos está indicando es que sólo podemos utilizar `await` dentro de una función `async`, es decir para poder bloquear el código con `await` se debe

utilizar una función en la que indiquemos que vamos a utilizar peticiones asíncronas.

Para resolver este problema deberemos incluir este código en una función async.

```
let url='http://localhost:3000/articulos'

async function getArticulos(){
  const response=await fetch(url)
  const articulos=await response.json();
  console.log(articulos)
}
```

De esta manera ya podemos realizar nuestra petición y gestionar los datos recibidos.

Si quisiéramos acceder a un artículo en concreto

```
let url='http://localhost:3000/articulos'

async function getArticulo(id){
  const response=await fetch(url + "/" + id)
  const articulo=await response.json();
  console.log(articulo)
}
```

Para la petición

```
getArticulo(1)
```

Nos devolverá

```
▶ {id: '1', nombre: 'Portátil Acer Aspire 5', precio: 700} async\_await.html:22
```

Para la petición

```
getArticulo(9)
```

Nos devolverá

```
✖ ▶ GET http://localhost:3000/articulos/9 404 async\_await.html:20 (Not Found)
```

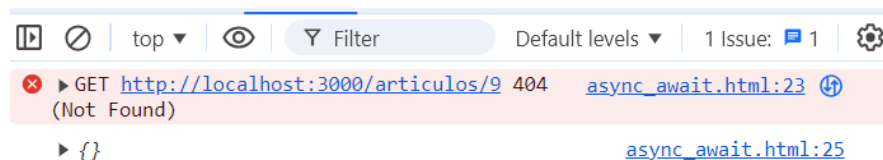
Como era de esperar, no existe el artículo con id 9 y se genera un error 404 que nos salta en la consola, pero que no hemos podido gestionar desde el código.

Para poder gestionar los errores con async/await deberemos utilizar el bloque try/catch

En nuestro caso modificamos la función para gestionar el error

```
async function getArticulo(id){
  try{
    const response=await fetch(url + "/" + id)
    const articulo=await response.json();
    console.log(articulo)
  }catch(error){
    console.log(error)
  }
}
```

Si volvemos a lanzar la petición para el artículo con id 9 obtenemos



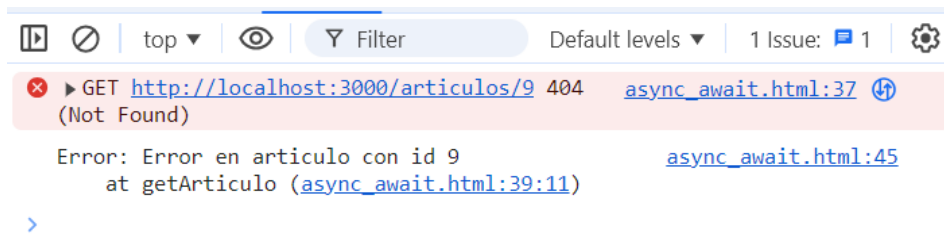
Nos aparece el error de la consola, pero nos trata de imprimir el objeto que debía haber recuperado y que no existe, es decir no está ejecutando nuestro error. Realmente lo está haciendo bien, porque no se ha producido ningún error en la petición, como vimos, fetch siempre devuelve una respuesta a la petición, para indicar que se ha producido un error debemos de comprobar la propiedad ok de la respuesta.

Modificamos nuestra función

```
async function getArticulo(id){
  try{
    const response=await fetch(url + "/" + id)
    if (!response.ok)
      throw new Error (`Error en articulo con id ${id}`)

    const articulos=await response.json();
    console.log(articulos)
  }
  catch(error){
    console.log(error)
  }
}
```


Si volvemos a lanzar la petición para el artículo con id 9



Ahora ya podemos gestionar nuestro error en la petición.

Vamos a ver cómo quedaría nuestra petición encadenada para obtener los datos del artículo que vende un proveedor

```
async function getArticuloProveedor(){
  let url='http://localhost:3000/'
  let idProveedor=prompt("Dime el id del proveedor: ")
  try{
    // Petición para proveedor
    let response=await fetch(url + "proveedores/" + idProveedor)
    if (!response.ok)
      throw new Error (`Error en proveedor con id ${idProveedor}` )
    let proveedor=await response.json();

    // Petición para artículo
    response=await fetch(url + "articulos/" + proveedor.idArticulo)
    if (!response.ok)
      throw new Error (`Error en artículo con id
${proveedor.idArticulo}` )
    let articulo=await response.json();
    console.log(articulo)
  }
  catch(error){
    console.log(error)
  }
}
```