# API Rest con JavaScript

Básicamente, una API Rest, es una interfaz que nos permite comunicarnos con otra aplicación que espera que hagamos una petición, puede ser para obtener datos, añadir datos, borrar…

# Un Ejemplo

[PokéApi](https://pokeapi.co/) es una API de prueba, nos proporciona varias rutas como esta <https://pokeapi.co/api/v2/pokemon/ditto> La cual nos proporciona los datos del pokemon Ditto.

Puedes probar esta ruta simplemente desde tu navegador. Con ella obtenemos un JSON con todos los datos del pokemon.

## Existen cuatro formas de invocar a una API Rest con JavaScript

### **1. XMLHttpRequest (AJAX)**

La primera forma de obtener datos que vamos a ver, va a ser mediante [XMLHttpRequest](https://developer.mozilla.org/es/docs/Web/API/XMLHttpRequest), un objeto de JavaScript que fue diseñado por Microsoft y adoptado por Mozilla, Apple y Google. Esta era una forma de programar usando un conjunto de tecnologías que te permitían crear páginas más dinámicas, al hacer peticiones al backend para obtener datos nuevos sin tener que recargar la página al completo.

El objeto XMLHttpRequest se creó originalmente para realizar peticiones HTTP a ficheros .xml externos desde Javascript. Actualmente, se realizan las mismas operaciones pero con ficheros [JSON](https://lenguajejs.com/javascript/objetos/json/), ya que son mucho más habituales en el ecosistema Javascript como **almacenamiento ligero de datos**.

El primer paso es crear un objeto XMLHttpRequest

// Creamos un nuevo XMLHttpRequest

const xhttp = new XMLHttpRequest();

Mediante esta instancia que acabamos de crear, podremos definir el tipo de petición utilizando el método open(method, url), indicando el método HTTP y la URL a la que queremos hacer la petición:

//Endpoint de la API y método que se va a usar para llamar

xhttp.open("GET", "https://pokeapi.co/api/v2/pokemon");  // Preparación de petición

xhttp.send();    //enviamos la petición

Con el método .open() solo preparamos la petición, pero la haremos finalmente efectiva con el método .send(), el cuál lanzará la petición al servidor. Poco después, si consultamos el contenido de la propiedad .responseText (texto de respuesta) de nuestra instancia, comprobaremos que debería estar el contenido del fichero que hemos pedido:

console.log(xhttp.responseText);

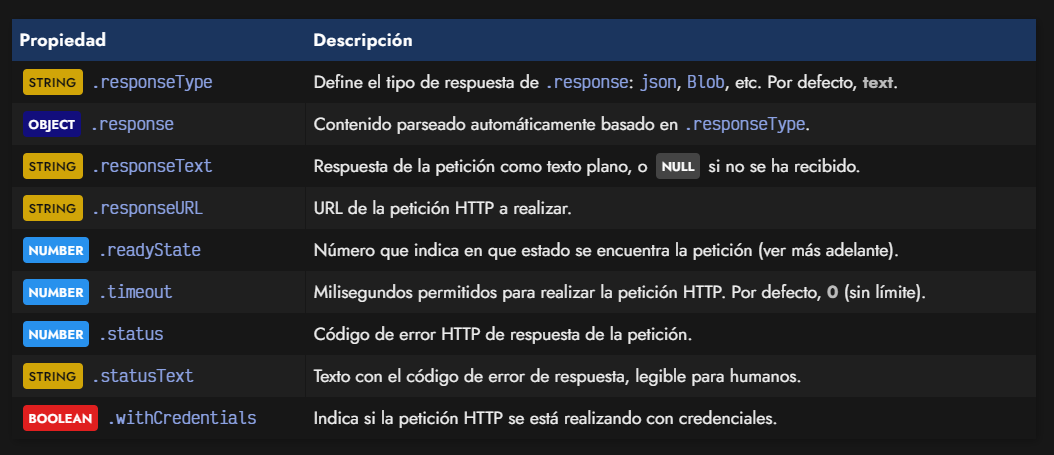
Este sería el funcionamiento base de las **peticiones XHR**, que como podes ver, es muy sencillo. Sin embargo, esto es sólo la teoría. A la hora de la práctica hay más detalles que hay que tener en cuenta.

Si intentamos ejecutar **todo el código** , no funcionará; el valor de xhttp.responseText nos aparecerá vacío.

Esto ocurre debido a que no estamos dando tiempo suficiente al navegador para realizar la petición al recurso al servidor, y por lo tanto, consultamos el resultado antes de tenerlo. Para solucionar esto, hay que esperar un tiempo para comprobar el valor, o mejor aún, hacerlo de forma **asíncrona**, como veremos más adelante.

### [El objeto XMLHttpRequest](https://lenguajejs.com/javascript/peticiones-http/xhr/#el-objeto-xmlhttprequest)

Al crear un objeto XMLHttpRequest tenemos acceso a una serie de propiedades que forman parte de nuestra instancia. Entre ellas, se encuentran las siguientes:



En primer lugar, las propiedades .responseType y .response se usan conjuntamente, debido a que si establecemos un valor en .responseType antes de hacer la petición, se procesará automáticamente la respuesta en .response. Por ejemplo, si establecemos el valor "json" en la propiedad .responseType, en .response obtendremos el objeto JSON parseado.

Por otro lado, en la propiedad .responseText siempre obtendremos el valor de texto puro, por si ese proceso de parseo queremos hacerlo manualmente. En la propiedad .responseURL tendremos la ruta completa de la petición HTTP.

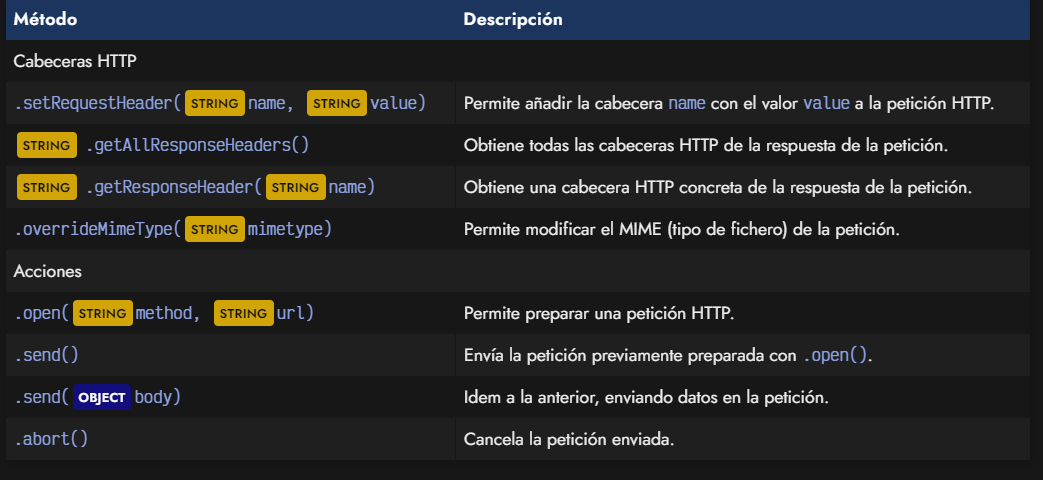
También tenemos la propiedad .readyState, en la cuál profundizaremos más adelante. Básicamente nos devuelve un valor que determina **en que punto se encuentra la petición** (iniciada, enviada, procesándose, finalizada). Es muy útil para saber cuando realizar ciertas acciones.

La propiedad .timeout indica el número de **milisegundos máximo** que permitiremos para realizar la petición. De ser superior a este tiempo, la petición se descartará. Por defecto viene establecido a 0, que significa que no hay límite de tiempo para la petición.

Las propiedades .status y .statusText nos muestran el **código de error HTTP** devuelto por la petición. La propiedad .status de vuelve el número en cuestión (200 = OK, 404 = Not Found, etc...) y la propiedad .statusText nos devuelve un texto un poco más legible donde nos da algo más de información.

Por último, el booleano .withCredentials nos indica si la petición se ha realizado enviando un usuario y contraseña o simplemente se trata de una petición sin indicar credenciales.

### [Métodos de XMLHttpRequest](https://lenguajejs.com/javascript/peticiones-http/xhr/#m%C3%A9todos-de-xmlhttprequest)



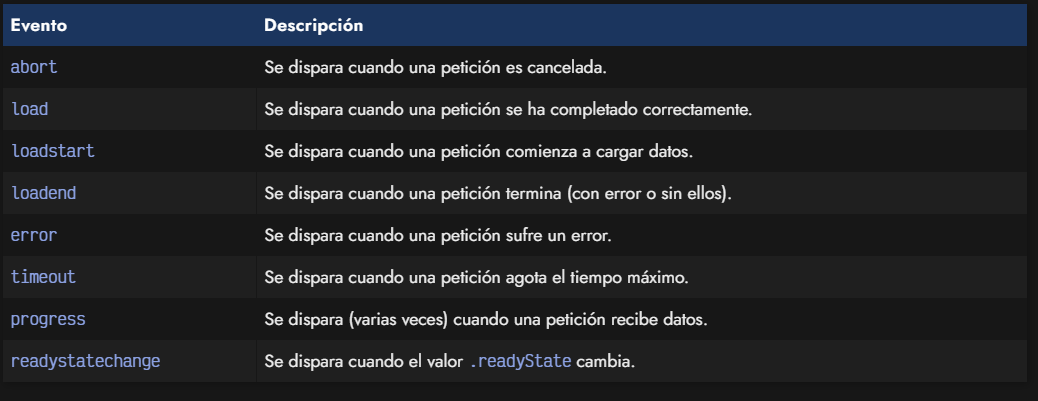
Los cuatro primeros se utilizan para temas relacionados con las **cabeceras de la petición HTTP**. El primero para la petición que enviamos (permite añadir cabeceras) y los dos siguientes para la petición de respuesta que recibimos, tanto obtener todas las cabeceras con .getAllResponseHeaders(), como obtener una específica .getResponseHeader(name).

El método .overrideMimeType() nos permite modificar (sobreescribir) el [MIME](https://es.wikipedia.org/wiki/Tipo_de_medio) de la petición recibido desde el servidor, muy útil cuando queremos forzarlo a uno diferente.

Por último, tenemos los métodos .open(), .send() y .abort() utilizados para iniciar, enviar y cancelar la petición respectivamente. En el caso de .send() es posible indicar un parámetro opcional que sea un objeto para enviar junto a la petición.

### [Eventos de XMLHttpRequest](https://lenguajejs.com/javascript/peticiones-http/xhr/#eventos-de-xmlhttprequest)

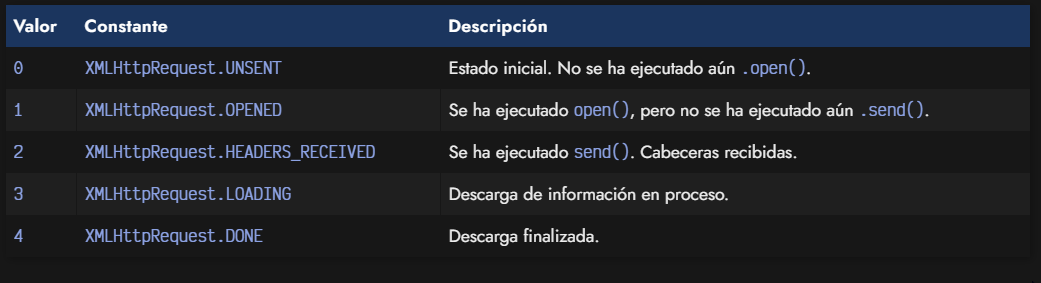
Durante el manejo de objetos XMLHttpRequest pueden suceder determinados eventos que podemos controlar desde nuestro código. Dichos eventos son los siguientes:



Los eventos del objeto XMLHttpRequest son bastante autodescriptivos. Quizás, el más complejo sea el evento readystatechange, el cuál tiene relación directa con la propiedad .readyState, que explicaremos detalladamente en el siguiente apartado.

### [El ciclo **readyState**](https://lenguajejs.com/javascript/peticiones-http/xhr/#el-ciclo-readystate)

La propiedad .readyState de nuestra instancia XMLHttpRequest  es un valor numérico (representado por una constante) que indica en el punto de progreso en que se encuentra la petición HTTP.



Para verlo más claramente, podemos observar el siguiente código, donde a través de la escucha del evento readystatechange de la instancia en cuestión, podemos estar pendientes de cuando obtenemos la información (readyState igual a 4 y status 200 OK):

const client = new XMLHttpRequest();

client.addEventListener("readystatechange", () => {

  const isDone = client.readyState === 4;

  const isOk = client.status === 200;

  if (isDone && isOk) {

    console.log(client.responseText);

  }

});

client.open("GET", "https://pokeapi.co/api/v2/pokemon");

client.send();

# Ejercicio Propuesto AJAX Javascript

1. Crea una llamada AJAX para recoger el contenido de un fichero JSON y mostrarlo por consola

datos.json

[

    {

        "nombre": "Ada",

        "apellido": "Lovelace",

        "profesion": "Programadora"

    },

    {

        "nombre": "Switch",

        "apellido": "Cisco",

        "profesion": "Experto en Redes"

    },

    {

        "nombre": "Tomcat",

        "apellido": "Apache",

        "profesion": "Experto en servidores"

    }

]

### **2. Fetch**

[Fetch](https://developer.mozilla.org/es/docs/Web/API/Fetch_API) es una nueva implementación de JavaScript, nos permite hacer lo mismo que **XMLHttpRequest** pero de manera más sencilla, el único defecto que le puedo ver es que al ser tan nueva, no funciona en internet Explorer, pero bueno, esto es un problema soportable para casi cualquier proyecto moderno.

### [Peticiones HTTP con fetch](https://lenguajejs.com/javascript/peticiones-http/fetch/#peticiones-http-con-fetch)

**Fetch** es el nombre de una nueva API para Javascript con la cuál podemos realizar peticiones HTTP asíncronas utilizando promesas y de forma que el código sea un poco más sencillo y menos verbose. La forma de realizar una petición es muy sencilla, básicamente se trata de llamar a fetch y pasarle por parámetro la URL de la petición a realizar:

const promise = fetch("https://pokeapi.co/api/v2/pokemon");

promise.then(function(response) {

  /\* ... \*/

});

El fetch() devolverá una **PROMISE** que será aceptada cuando reciba una respuesta y sólo será rechazada si hay un fallo de red o si por alguna razón no se pudo completar la petición.

Al método .then() se le pasa una función callback donde su parámetro response es el objeto de respuesta de la petición que hemos realizado. En su interior realizaremos la lógica que queramos hacer con la respuesta a nuestra petición.

Esto se suele reescribir de la siguiente forma, que queda mucho más simple y evitamos constantes o variables temporales de un solo uso:

fetch("https://pokeapi.co/api/v2/pokemon")

  .then(function(response) {

    /\*\* Código que procesa la respuesta \*\*/

  });

### [Opciones de **fetch()**](https://lenguajejs.com/javascript/peticiones-http/fetch/#opciones-de-fetch)

A la función fetch(), al margen de la url a la que hacemos petición, se le puede pasar un segundo parámetro de opciones de forma opcional, un **OBJECT** con opciones de la petición HTTP:

const options = {

  method: "GET"

};

fetch("https://pokeapi.co/api/v2/pokemon", options)

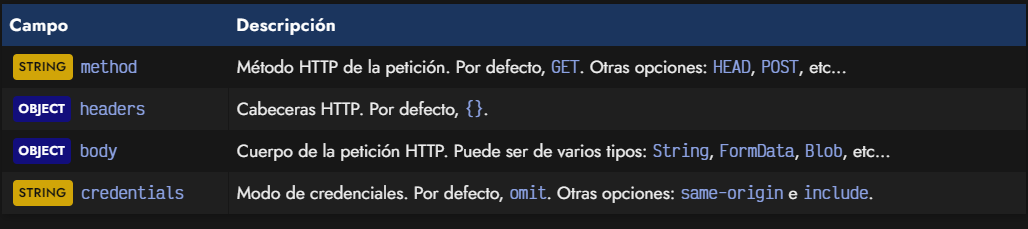
  .then(response => response.text())

  .then(data => {

    /\*\* Procesar los datos \*\*/

  });

Un poco más adelante, veremos como trabajar con la respuesta response, pero vamos a centrarnos ahora en el parámetro opcional options de la petición HTTP. En este objeto podemos definir varios detalles:



Lo primero, y más habitual, suele ser indicar el método HTTP a realizar en la petición. Por defecto, se realizará un GET, pero podemos cambiarlos a HEAD, POST, PUT o cualquier otro tipo de método. En segundo lugar, podemos indicar objetos para enviar en el body de la petición, así como modificar las cabeceras en el campo headers:

const options = {

  method: "POST",

  headers: {

    "Content-Type": "application/json"

  },

  body: JSON.stringify(jsonData)

};

En este ejemplo, estamos enviando una petición POST, indicando en la cabecera que se envía contenido JSON y en el cuerpo de los datos, enviando el objeto jsonData, codificado como texto mediante stringify().

Por último, el campo credentials permite modificar el modo en el que se realiza la petición. Por defecto, el valor omit hace que no se incluyan credenciales en la petición, pero es posible indicar los valores same-origin, que incluye las credenciales si estamos sobre el mismo dominio, o include que incluye las credenciales incluso en peticiones a otros dominios.

Recuerda que estamos realizando peticiones relativas al **mismo dominio**. En el caso de realizar peticiones a dominios diferentes obtendríamos un problema de CORS (Cross-Origin Resource Sharing) similar al siguiente:

Access to fetch at 'https://otherdomain.com/file.json' from origin 'https://domain.com/' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

Más adelante hablaremos de CORS y de como solucionar estos problemas si necesitamos realizar peticiones a otros dominios.

### [Cabeceras (Headers)](https://lenguajejs.com/javascript/peticiones-http/fetch/#cabeceras-headers)

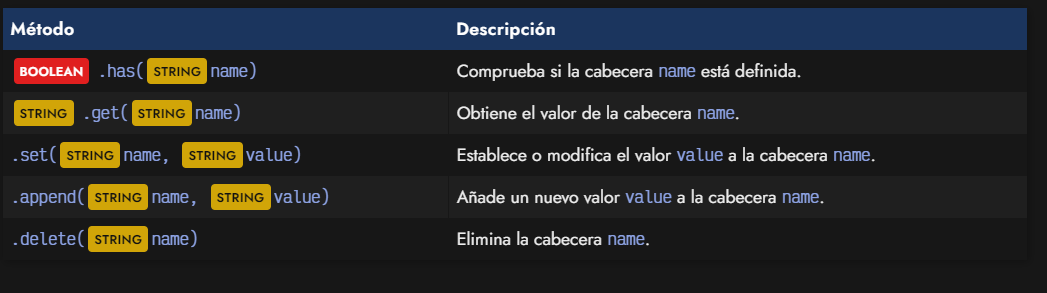
Aunque en el ejemplo anterior hemos creado las cabeceras como un **OBJECT** genérico de Javascript, es posible crear un objeto Headers con el que trabajar:

const headers = new Headers();

headers.set("Content-Type", "application/json");

headers.set("Content-Encoding", "br");

Para ello, a parte del método .set() podemos utilizar varios otros para trabajar con cabeceras, comprobar su existencia, obtener o cambiar los valores o incluso eliminarlos:



Como muchos otros objetos iterables, podemos utilizar los métodos .entries(), .keys() y/o .values() para recorrerlos:

for ([key, value] of headers.entries()) {

  console.log("Clave: ", key, "valor: ", value);

}

Para peticiones con pocas cabeceras no hay mayor problema, pero en peticiones más complejas utilizar Headers es una buena práctica.

### [Respuesta (Response)](https://lenguajejs.com/javascript/peticiones-http/fetch/#respuesta-response)

fetch("/robots.txt", options)

  .then(response => response.text())

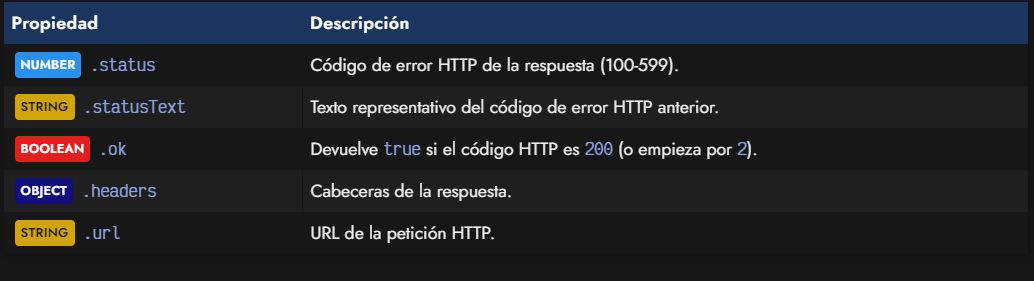
  .then(data => {

    /\*\* Procesar los datos \*\*/

  });

Aunque en este ejemplo, simplemente estamos utilizando una arrow function que hace un return implícito de la promesa que devuelve el método .text(), dicho objeto response tiene una serie de propiedades y métodos que pueden resultarnos útiles al implementar nuestro código.

Por el lado de las **propiedades**, tenemos las siguientes:



Si venimos de XMLHttpRequest, esto no nos resultará nada extraño. Las propiedades .status y statusText nos devuelven el **código de error HTTP** de la respuesta en formato numérico y cadena de texto respectivamente.

Sin embargo, existe una novedad respecto a XHR, y es que tenemos una propiedad .ok que nos devuelve true si el código de error de la respuesta es un valor del rango 2xx, es decir, que todo ha ido correctamente. Así pues, tenemos una forma práctica y sencilla de comprobar si todo ha ido bien al realizar la petición:

fetch("/robots.txt")

  .then(response => {

    if (response.ok)

      return response.text()

  })

Por último, tenemos la propiedad .headers que nos devuelve las cabeceras de la respuesta y la propiedad .url que nos devuelve la URL completa de la petición que hemos realizado.

### [Procesar la respuesta](https://lenguajejs.com/javascript/peticiones-http/fetch/#procesar-la-respuesta)

Por otra parte, la instancia response también tiene algunos **métodos** interesantes, la mayoría de ellos para procesar mediante una promesa los datos recibidos y facilitar el trabajo con ellos:



Observa que en los ejemplos anteriores hemos utilizado response.text(). Este método indica que queremos procesar la respuesta como datos textuales, por lo que dicho método devolverá una **PROMISE** con los datos en texto plano, facilitando trabajar con ellos de forma manual:

fetch("/robots.txt")

  .then(response => response.text())

  .then(data => console.log(data));

Observa que en este fragmento de código, tras procesar la respuesta con response.text(), devolvemos una  con el contenido en texto plano. Esta **PROMISE**  se procesa en el segundo .then(), donde gestionamos dicho contenido almacenado en data.

Tener en cuenta que tenemos varios métodos similares para procesar las respuestas. Por ejemplo, el caso anterior utilizando el método response.json() en lugar de response.text() sería equivalente al siguiente fragmento:

fetch("/contents.json")

  .then(response => response.text())

  .then(data => {

    const json = JSON.parse(data);

    console.log(json);

  });

Como se puede ver, con response.json() nos ahorraríamos tener que hacer el JSON.parse() de forma manual, por lo que el código es algo más directo.

### [Fetch usando **.then()**](https://lenguajejs.com/javascript/peticiones-http/fetch/#fetch-usando-then)

Ejemplo

fetch('https://pokeapi.co/api/v2/pokemon')

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

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

      .catch(err => console.error('Error', err.message));

Lo que vemos a continuación sería un ejemplo un poco más completo de todo lo que hemos visto hasta ahora:

* Comprobamos que la petición es correcta con response.ok
* Utilizamos response.text() para procesarla
* En el caso de producirse algún error, lanzamos excepción con el código de error
* Procesamos los datos y los mostramos en la consola
* En el caso de que la **PROMISE**  sea rechazada, capturamos el error con el catch
* Si ocurre un error 404, 500 o similar, lanzamos error con throw para capturarlo en el catch

fetch("/robots.txt")

  .then(response => {

    if (response.ok)

      return response.text();

    throw new Error(response.status);

  })

  .then(data => {

    console.log("Datos: " + data);

  })

  .catch(err => {

    console.error("ERROR: ", err.message)

  });

Podemos refactorizar un poco este ejemplo para hacerlo más legible. Creamos la función isResponseOk() para procesar la respuesta (invirtiendo el condicional para hacerlo más directo). Luego, los .then() y .catch() los utilizamos con una arrow function para simplificarlos:

const isResponseOk = (response) => {

  if (!response.ok)

    throw new Error(response.status);

  return response.text()

}

fetch("/robots.txt")

  .then(response => isResponseOk(response))

  .then(data => console.log("Datos: ", data))

  .catch(err => console.error("ERROR: ", err.message));

Sin embargo, aunque es bastante común trabajar con promesas utilizando .then(), también podemos hacer uso de async/await para manejar promesas, de una forma un poco más directa. La única diferencia es que con .then() el código no es bloqueante, mientras que con async/await si es bloqueante.

### [Fetch usando **async/await**](https://lenguajejs.com/javascript/peticiones-http/fetch/#fetch-usando-asyncawait)

Utilizar async/await no es más que lo que se denomina **azúcar sintáctico**, es decir, utilizar algo visualmente más agradable, pero que por debajo realiza la misma tarea. Para ello, lo que debemos tener siempre presente es que un await sólo se puede ejecutar si esta dentro de una función definida como async.

En este caso, lo que hacemos es lo siguiente:

* Creamos una función request(url) que definimos con async
* Llamamos a fetch utilizando await para esperar y resolver la promesa
* Comprobamos si todo ha ido bien usando response.ok
* Llamamos a response.text() utilizando await y devolvemos el resultado

const request = async (url) => {

  const response = await fetch(url);

  if (!response.ok)

    throw new Error("WARN", response.status);

  const data = await response.text();

  return data;

}

const resultOk = await request("/robots.txt");

const resultError = await request("/nonExistentFile.txt");

Una vez hecho esto, podemos llamar a nuestra función request y almacenar el resultado, usando nuevamente await.

### **3. Axios**

Hasta ahora, tanto **XMLHttpRequest** como **Fetch** eran nativas de JavaScript, pero [Axios](https://github.com/axios/axios) es una librería externa que tendremos que importar en nuestro proyecto antes de usarla.

<head>

  <script src="https://unpkg.com/axios/dist/axios.min.js"></script>

</head>

Una vez importada, podrás usarla en tu proyecto.

axios.get('https://pokeapi.co/api/v2/pokemon')

.then(function (response) {

  // función que se ejecutará al recibir una respuesta

  console.log(response);

})

.catch(function (error) {

  // función para capturar el error

  console.log(error);

})

.then(function () {

  // función que siempre se ejecuta

});

#### ¿Por qué se usa tanto Axios? Parece que Fetch es perfecto ¿para que iba a querer importar otra librería a mi proyecto?

El primer motivo es sencillo, Fetch es nuevo y no puede implementarse en proyectos que sigan usando tecnologías viejas, algunas veces está limitado, mientras que Axios tiene muy buena compatibilidad.

Otro motivo muy importante es que Axios parsea automáticamente las respuestas JSON.

// axios

axios.get('https://pokeapi.co/api/v2/pokemon')

  .then(response => {

    console.log(response.data); // response.data ya es un JSON

  }, error => {

    console.log(error);

  });

// fetch()

fetch('https://pokeapi.co/api/v2/pokemon')

  .then(response => response.json())    // a fetch le llega una respuesta en string que tiene que ser parseada a JSON

  .then(data => {

    console.log(data)

  })

  .catch(error => console.error(error));

### **4. jQuery.Ajax()**

Obviamente si queremos hacer este tipo de llamadas, tendremos que importar la librería de [jQuery](https://jquery.com/) a nuestro proyecto.

jQuery es una librería que muchos dan por muerta, pero creo que todavía le queda mucho recorrido, no hay más que ver las encuestas de [StackOverflow](https://insights.stackoverflow.com/survey/2020" \l "most-popular-technologies) o [StateOfJs](https://2020.stateofjs.com/en-US/other-tools/), sigue siendo una de las librerías/frameworks más usados, por lo que creo que es necesario conocer esta manera de llamar a una API Rest con jquery.

<head>

  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>

</head>

$.ajax({

  type: 'GET',

  url: 'https://pokeapi.co/api/v2/pokemon',

  dataType: 'json',

  success: function(data) {

    console.log(data)

  }

});