



flex, grid...

CTRL+K

# Fetch: Peticiones Asíncronas

Mecanismo moderno para hacer peticiones asíncronas

Una vez que aprendemos a realizar [peticiones HTTP mediante XHR](#) nos damos cuenta que es un mecanismo muy interesante y útil, y que nos abre un mundo de posibilidades trabajando con Javascript. Sin embargo, con el tiempo nos vamos dando cuenta también, que la forma de trabajar con objetos `XMLHttpRequest`, aunque es muy potente requiere mucho trabajo que hace que el código no sea tan legible y práctico como quizás debería.

Entonces es cuando surge `fetch`, un sistema más moderno, basado en promesas de Javascript, para realizar peticiones HTTP asíncronas de una forma más legible y cómoda.

## Peticiones con el método `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:

JS

```
const promise = fetch("/robots.txt");

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.

El modo más habitual de manejar las promesas es utilizando `.then()`, aunque también se puede utilizar [async/await](#). Esto se suele reescribir de la siguiente forma, que queda mucho más simple y evitamos constantes o variables temporales de un solo uso:

JS

```
fetch("/robots.txt")
  .then(function(response) {
    /** Código que procesa la respuesta **/
  });
```

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.

Opciones   Credentials   Headers   Response

## ↳ 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:

JS

```
const options = {
  method: "GET"
};

fetch("/robots.txt", 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:

Campo
Descripción
<code>method</code>
Método HTTP de la petición. Por defecto, <code>GET</code> . Otras opciones: <code>HEAD</code> , <code>POST</code> , etc...
<code>headers</code>
Cabeceras HTTP. Por defecto, <code>{}</code> .
<code>body</code>

Campo
Descripción
Cuerpo de la petición HTTP. Puede ser de varios tipos: <code>String</code> , <code>FormData</code> , <code>Blob</code> , etc...
<code>s credentials</code>
Modo de credenciales. Por defecto, <code>omit</code> . Otras opciones: <code>same-origin</code> e <code>include</code> .

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`:

JS

```
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()`.

## Procesando 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:

Método
Descripción
<code>s .text()</code>
Devuelve una promesa con el texto plano de la respuesta.
<code>o .json()</code>
Idem, pero con un objeto <code>json</code> . Equivale a usar <code>JSON.parse()</code> .
<code>o .blob()</code>

Método	Descripción
	Idem, pero con un objeto <code>Blob</code> (binary large object).
◦ <code>.arrayBuffer()</code>	Idem, pero con un objeto <code>ArrayBuffer</code> (buffer binario puro).
◦ <code>.formData()</code>	Idem, pero con un objeto <code>FormData</code> (datos de formulario).
◦ <code>.clone()</code>	Crea y devuelve un clon de la instancia en cuestión.
◦ <code>Response.error()</code>	Devuelve un nuevo objeto <code>Response</code> con un error de red asociado.
◦ <code>Response.redirect(url, code)</code>	Redirige a una <code>url</code> , opcionalmente con un <code>code</code> de error.

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:

JS

```
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 `PROMISE` con el contenido en texto plano. Esta `PROMISE` se procesa en el segundo `.then()`, donde gestionamos dicho contenido almacenado en `data`.

Ten 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:

JS

```
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.

# Consumiendo la promesa

Tenemos dos formas principales de consumir la promesa.

Promesas vía `then()`   Promesas vía `async/await`

---

## ↳ Promesas usando `.then()`

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`

JS

```
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.

► Más información: [Promesas con .then\(\)](#) (*thenables, no bloqueantes*)



Si te interesa profundizar más en todo este tema de promesas, deberías echarle un vistazo al [minicurso de Asincronía Javascript](#).

**AJAX: Peticiones HTTP**  
Capítulo anterior

**XHR: XMLHttpRequest**  
Capítulo siguiente

**Volver**  
Al índice

**Acceder a Discord**  
Comunidad de Manz.dev

## RELACIONADOS

En mis canales de Youtube [@ManzDev](#) y [ManzDevTy](#), tienes más contenido...








## APRENDER MÁS

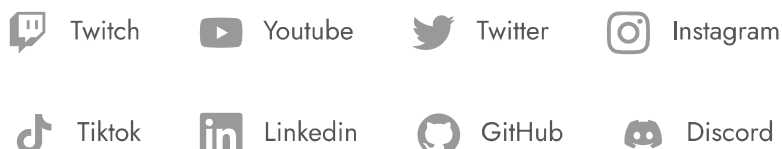
Si lo prefieres, puedes aprender también sobre estas temáticas:



## ¿QUIÉN SOY YO?

Soy Manz, vivo en Tenerife (España) y soy streamer partner en Twitch  y profesor. Me apasiona el universo de la programación web, el diseño y desarrollo web y la tecnología en general. Aunque soy full-stack, mi pasión es el front-end, la terminal y crear cosas divertidas y locas.

Puedes encontrar más sobre mi en [Manz.dev](https://Manz.dev)



Creado por Manz con  Alojado en [DigitalOcean](https://DigitalOcean.com).  
© Todos los derechos reservados. Los izquierdos también.