API Fetch de JavaScript: Qué es y cómo se usa

neoguias.com/fetch-javascript

por Edu Lázaro | @neeonez

3 de julio de 2020

En este tutorial veremos lo que es la API Feth de JavaScript, mediante la cual podemos realizar peticiones asíncronas mediante promesas, facilitando mucho ciertas tareas para las que antaño necesitábamos usar alguna librería.

Qué es la API Fetch

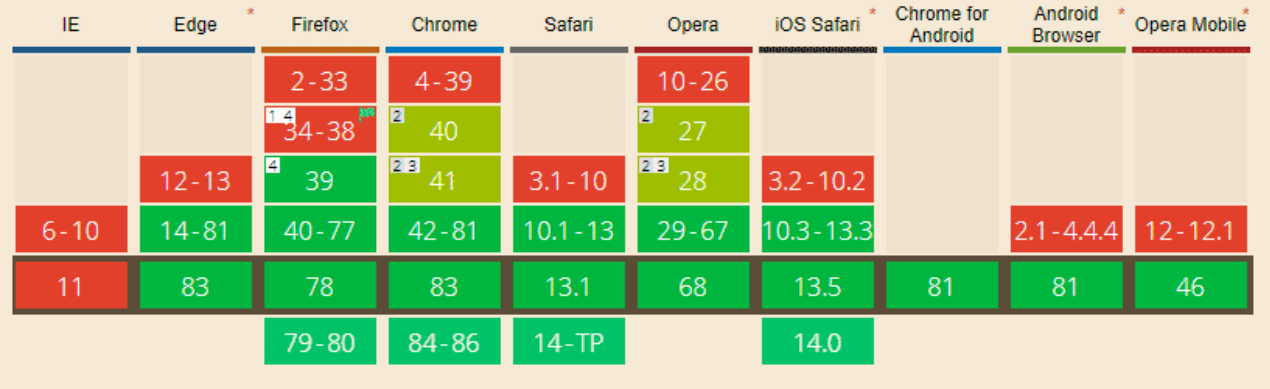
La API Fetch es una funcionalidad relativamente nueva que se incluye en las últimas versiones de JavaScript mediante la cual podemos realizar peticiones asíncronas y obtener respuesta a las mismas de una forma sencilla.

Desde finales de los 90 era ya posible crear peticiones de red asíncronas en varios navegadores, como por ejemplo Internet Explorer 5. Para ello se usaban peticiones **XMLHttpRequest**, también conocidas como **XHR**. No tardaron en aparecer las primeras aplicaciones que sacaban partido a estas peticiones, como fue el caso de Gmail, entre muchas otras. Finalmente, debido a la popularidad de estas peticiones, la tecnología recibió el nombre de **AJAX: Asynchronous JavaScript And XML**.

Sin embargo, la gestión de este tipo de peticiones era muy compleja, suponiendo verdaderos quebraderos de cabeza, especialmente debido a los diferentes métodos que cada navegador usaba para gestionar estas peticiones. Debido a ello, era habitual usar librerías que abstraían el código de las peticiones XHR en una serie de sencillas funciones. En concreto, la librería más utilizada fue jQuery, que incluía el método `jQuery.ajax()`, así como métodos que simplificaban todavía más ciertos tipos de petición como `jQuery.get()` o `jQuery.post()`.

La librería jQuery to tardó en hacerse tan famosa que era usada por la gran mayoría de las aplicaciones web y, de hecho, todavía sigue siendo muy utilizada.

Sin embargo, la API Fetch ha tomado el relevo de la gestión de peticiones asíncronas, usando funciones promesas para su creación.



Además, esta API es ampliamente soportada por los navegadores actuales, a excepción de las versiones más antiguas de Internet Explorer. Sin embargo, existen librerías como esta que permiten usar la función `fetch` en cualquier navegador existente.

Cómo usar Fetch

El uso de la API Fetch es muy sencillo. Vamos a comenzar creando una petición asíncrona GET mediante la cual vamos a obtener los datos de un archivo:

```
fetch('/archivo.json');
```

Y con esto la petición HTTP se realizará automáticamente. La única condición es que el archivo o la ruta que quieres obtener esté en el mismo dominio. La función `fetch` es una función global que está disponible en el objeto `window` del navegador.

A continuación vamos a mostrar por la consola los datos del archivo que hemos obtenido, para lo cual usaremos una serie de promesas:

```
fetch('./archivo.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

Como ves, la función `fetch()` devuelve una promesa, teniendo que esperar a que esta se resuelva, momento en el que se ejecutará la función definida en el método `then()` de la misma. La primera función o handler recibe como parámetro el resultado obtenido de la promesa `fetch`, que es un objeto de tipo **Response**. En breve veremos este objeto en detalle.

Como ya hemos visto, la función `fetch` acepta dos parámetros; uno con la URL a la que se debe acceder y otro con las opciones. Devuelve una promesa, por lo que tendremos que usar los métodos `then`, `catch` o `finally` según nos convenga. Si no conoces las promesas, consulta el siguiente tutorial, en donde explico qué son y cómo se utilizan las promesas.

En el siguiente ejemplo obtendremos los datos de un usuario de GitHub usando `fetch` y mostraremos el resultado por pantalla:

```
fetch('https://api.github.com/users/edulazaro')
  .then(response => {
    return response.json();
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error(error);
  });
```

En el código anterior realizamos las siguientes tareas.

1. Primero, la función `fetch` realizará una petición a la URL `https://api.github.com/users/edulazaro`, que esperará asíncronamente a que se reciba una respuesta.
2. En el primer `then` pasamos el resultado obtenido a una función anónima que formatea la respuesta como JSON pasando la respuesta al segundo `then`.
3. El segundo `then` muestra el código JSON por la consola.
4. Finalmente, usamos el método `catch` para mostrar un posible error en la consola en caso de que se produzca, tal y como veremos en el siguiente apartado.

Este será el resultado que se mostrará por la consola, que son los datos del usuario de GitHub `edulazaro` en formato JSON:

```
avatar_url: "https://avatars2.githubusercontent.com/u/7797530?v=4"
bio: "I'm quite proud of instruction at referenced memory could not be read,
honestly."
blog: "https://www.neoguias.com"
company: null
created_at: "2014-06-04T18:46:12Z"
email: null
events_url: "https://api.github.com/users/edulazaro/events{/privacy}"
followers: 5
followers_url: "https://api.github.com/users/edulazaro/followers"
following: 4
following_url: "https://api.github.com/users/edulazaro/following{/other_user}"
gists_url: "https://api.github.com/users/edulazaro/gists{/gist_id}"
gravatar_id: ""
hireable: true
html_url: "https://github.com/edulazaro"
id: 7797530
location: "London"
login: "edulazaro"
name: "Eduardo Lázaro"
node_id: "MDQ6VXNlcjc3OTc1MzA="
organizations_url: "https://api.github.com/users/edulazaro/orgs"
public_gists: 0
public_repos: 15
received_events_url: "https://api.github.com/users/edulazaro/received_events"
repos_url: "https://api.github.com/users/edulazaro/repos"
site_admin: false
starred_url: "https://api.github.com/users/edulazaro/starred{/owner}{/repo}"
subscriptions_url: "https://api.github.com/users/edulazaro/subscriptions"
twitter_username: null
type: "User"
updated_at: "2021-01-11T15:36:55Z"
url: "https://api.github.com/users/edulazaro"
```

Gestión de errores

La función `fetch()` devuelve una promesa, por lo que siempre podemos usar el método `catch()` de la misma para detectar cualquier posible error que haya ocurrido durante la petición o el procesamiento de la misma. El método `catch()` recibirá el error que ha ocurrido como parámetros. En el siguiente ejemplo mostramos dicho error por la consola:

```
fetch('./archivo.json')
  .then(response => response.json())
  .then(data => console.log(data));
  .catch(error => console.error(error));
```

También podemos gestionar los errores de otra forma. Por ejemplo, aunque no es lo habitual, podemos detectar el error en la respuesta que obtenemos en el primer `then()`:

```
fetch('./archivo.json')
.then(response => {
  if (!response.ok) {
    // Lanzamos el error
    throw Error(response.statusText);
  }
  return response;
})
.then(response => {
  //No ha ocurrido ningún error
});
```

Tipos de petición

La función `fetch` acepta un segundo parámetro mediante el cual podemos personalizar la petición, pudiendo crear peticiones GET, POST, PUT, PATCH, OPTIONS o DELETE entre otros métodos HTTP. Para ello debes especificar el método en la propiedad `method` de este segundo objeto. Además también puedes configurar las cabeceras mediante la propiedad `headers` y el body mediante la propiedad `body` .

En este ejemplo enviamos una petición POST:

```
const peticion= {
  method: 'POST',
  headers: {
    'Content-type': 'application/x-www-form-urlencoded; charset=UTF-8'
  },
  body: 'marca=Fiat&modelo=Punto'
}

fetch('crear-modelo', peticion).catch(error => {
  console.error('Ha ocurrido un error', error);
});
```

También podemos crear una petición POST en la que enviemos datos JSON:

```
const peticion = {
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({marca: 'Fiat', modelo: 'Punto'})
}

fetch('/crear-modelo', peticion).catch(error => {
  console.error('Ha ocurrido un error', error);
});
```

Para entender la API Fetch necesitas entender también cómo funcionan los objetos **Response** y **Request**, que es lo que veremos a continuación.

El objeto Response

Este objeto es el que devuelven las peticiones `fetch()`, incluyendo toda la información acerca de la petición realizada y de la respuesta obtenida. Vamos a ver las diferentes propiedades de este objeto.

Contenido

Se trata del cuerpo o body que se obtiene como respuesta a la peticiones, que puede ser contenido HTML, XML, JSON o cualquier otro contenido válido. Puede acceder al body mediante estos métodos:

- `text()`: Devuelve el contenido como una cadena de texto plano.
- `json()`: Devuelve el contenido como un objeto JSON ya parseado.
- `blob()`: Devuelve el contenido como un objeto de tipo Blob.
- `formData()`: Devuelve el contenido como un objeto de tipo FormData.
- `arrayBuffer()`: Devuelve el contenido como un objeto de tipo ArrayBuffer.

Todos los métodos anteriores devolverán una promesa. Por ejemplo, si te interesase obtener solamente el texto de la respuesta usarías el método `response.text()`:

```
fetch('./archivo.json')
  .then(response => response.text())
  .then(contenido => console.log(contenido));
```

En caso de que quisieses obtener el contenido en formato JSON, usarías el método `response.json()`:

```
fetch('./archivo.json')
  .then(response => response.json())
  .then(contenido => console.log(contenido));
```

Además, desde ES8 también puedes usar las funciones **async/await** para obtener respuestas mediante `fetch`:

```
(async () => {
  const response = await fetch('./archivo');
  const data = await response.json();
  console.log(data);
})();
```

Metadatos

Además del contenido o body de la respuesta, también podrás acceder a una serie de metadatos que se incluyen como parte de la respuesta.

url

Mediante la propiedad `url` del objeto `response` podrás acceder a la URL de la petición que acabamos de realizar. Puedes obtenerla de este modo:

```
fetch('./archivo.json').then(response => { response =>
  console.log(response.url);
});
```

headers

Mediante la propiedad `headers` del objeto `response` podrás acceder a las diferentes cabeceras HTTP que se han obtenido como respuesta a la petición. Por ejemplo, vamos a obtener la cabecera `'Content-Type'`:

```
fetch('./archivo.json').then(response => {
  const contentType = response.headers.get('Content-Type');
  console.log(contentType);
});
```

Todo depende de lo que devuelva el servidor, pero es de esperar que se muestre `application/json` en la consola.

También puedes obtener otras cabeceras, como la fecha de la petición:

```
fetch('./archivo.json').then(response => {
  const fecha = response.headers.get('Date');
  console.log(fecha);
});
```

status

Mediante la propiedad `headers` del objeto `response` podrás ver estado de la respuesta HTTP que se ha obtenido como resultado de la petición. Aquí tienes un ejemplo:

```
fetch('./archivo.json').then(response => {
  response => console.log(response.status);
});
```

El estado está representado por un número entero:

- **Petición realizada correctamente**: Estados 200 a 299.
- **Respuesta informativa**: Estados 100 a 199.
- **Error en la petición (cliente)**: Estados 400 a 499.
- **Error en la petición (servidor)**: Estados 500 a 599.
- **Redirección**: Estados 300 a 399.

Si quieres, puedes consultar la lista completa de códigos de respuesta HTTP aquí.

statusText

Mediante la propiedad `statusText` del objeto `response` podrás acceder al mensaje de estado que se ha obtenido junto con la respuesta. Si todo ha ido bien, el mensaje será sencillamente `OK`, aunque es habitual personalizarlo en el servidor cuando se envía algún tipo de error. Puedes obtener la propiedad `statusText` de este modo:

```
fetch('./archivo.json').then(response => { response =>
  console.log(response.statusText);
});
```

El objeto Request

El objeto **Request** se usa para representar la petición que queremos realizar. Cuando usas la función `fetch`, en el fondo, siempre se crea un objeto de este tipo aunque no lo definas tú de forma explícita. Puedes crear una instancia de este objeto así:

```
const peticion = new Request('./archivo.json');
```

Mientras que hasta ahora habíamos creado las peticiones así:

```
fetch('./archivo.json')
  .then(response => response.json())
  .then(contenido => console.log(contenido));
```

Usando un objeto Request también podrás podrás crearlas así:

```
const peticion = new Request('./archivo.json');
fetch(peticion)
  .then(response => response.json())
  .then(contenido => console.log(contenido));
```

Este objeto incluye, entre otras, diversas propiedades de solo lectura que te permiten inspeccionar la petición:

- `url`: Contiene la URL de la petición.
- `method`: Contiene el método de la petición, ya sea GET, POST, PATCH o cualquier otro.
- `headers`: Contiene las cabeceras de la petición.
- `referrer`: Contiene la página o recurso desde la que se realiza la petición.
- `cache`: El tipo de caché de la petición, de haberlo, cuyo valor puede ser *default*, *reload* o *no-cache*, entre otros.

También podrás establecer las cabeceras y demás datos de la petición. Para agregar cabeceras, puedes crear una instancia del objeto `Headers`, tal que así:

```
const cabeceras = new Headers({
  'Content-Type': 'application/json'
});
```

También puedes crear primero el objeto y agregarlas más tarde:

```
const cabeceras = new Headers();
cabeceras.append('Content-Type', 'application/json');
```

También podrás comprobar si una cabecera existe en un objeto de tipo `Headers` mediante el método `has()`, u obtener el valor de una cabecera mediante el método `get()`:

```
if (cabeceras.has('Content-Type')) {
  const valor = cabeceras.get('Content-Type');
  console.log(valor);
}
```

Del mismo modo, también puedes eliminar una cebecera mediante el método `delete()`:

```
cabeceras.delete('Content-Type');
```

Luego, puedes agregar este objeto en el campo `headers` de la petición:

```
const peticion = new Request('./archivo.json', {
  headers: cabeceras
});
fetch(peticion);
```

Aunque también puedes crearlas directamente:

```
const peticion = new Request('./file.json', {
  headers: new Headers({
    'Content-Type': 'application/json'
  })
});
fetch(peticion);
```

Cancelar peticiones Fetch

Hasta hace no demasiado, no existía ninguna forma de terminar o abortar peticiones `fetch` una vez iniciadas. Sin embargo, ahora podemos usar `AbortController` y `AbortSignal`.

Para cancelar una petición tendrás que crear primero una señal creando una instancia de la clase `AbortController`:

```
const controller = new AbortController();
const signal = controller.signal;
```

A continuación vamos a establecer un timeout de modo que el método `abort()` del objeto `controller` se ejecute, por ejemplo, a los seis segundos:

```
setTimeout(() => controller.abort(), 6000);
```

Ahora bastará pasar con la señal como uno de los parámetros de configuración de la función `fetch`:

```
fetch('./archivo.json', { signal });
```

En caso de que no hayas obtenido respuesta a la petición al cabo de seis segundos, esta se cancelará. En caso de obtenerla, la petición se ejecutará normalmente.

Es posible diferenciar este error de cualquier otro que haya ocurrido; útil a la hora de gestionar los errores de la petición. Cuando la petición sea abortada, se producirá un error en la promesa de tipo `DOMException` que tendrá el nombre `AbortError`:

```
fetch('./archivo.json', { signal })
  .then(response => response.text())
  .then(contenido => console.log(contenido))
  .catch(error => {
    if (error.name === 'AbortError') {
      console.error('Petición abortada')
    } else {
      console.error('Cualquier otro error', error);
    }
  });
```

Alternativas a Fetch

Tienes diferentes alternativas a la API Fetch. Por ejemplo, es también común usar la librería Axios de JavaScript, que es preferida por muchos desarrolladores. También podrías usar la librería jQuery o crear peticiones directamente mediante el objeto `XMLHttpRequest`, aunque esto último resulta bastante tedioso.