

# SOLICITUDES DE RED: FETCH

1

*2º DAW - Desarrollo Web en Entorno Cliente*

# Solicitudes de Red (Fetch)

- JavaScript puede enviar peticiones de red al servidor y cargar nueva información siempre que se necesite. Por ejemplo, podemos utilizar una petición de red para:
  - Crear una orden,
  - Cargar información de usuario,
  - Recibir las últimas actualizaciones desde un servidor,
  - ...etc.
- ...Y todo esto sin la necesidad de refrescar la página.
- Se utiliza el término global “AJAX” (abreviado Asynchronous JavaScript And XML, en español: “JavaScript y XML Asíncrono”) para referirse a las peticiones de red originadas desde JavaScript. Sin embargo, no estamos necesariamente condicionados a utilizar XML dado que el término es antiguo y es por esto que el acrónimo XML se encuentra aquí. Probablemente lo hayáis visto anteriormente. Existen múltiples maneras de enviar peticiones de red y obtener información de un servidor.
- Comenzaremos con el método `fetch()` que es moderno y versátil. Este método no es soportado por navegadores antiguos (sin embargo se puede incluir un polyfill), pero es perfectamente soportado por los navegadores actuales y modernos. La sintaxis básica es la siguiente:

```
1 let promise = fetch(url, [options])
```

- `url` – representa la dirección URL a la que deseamos acceder.
- `options` – representa los parámetros opcionales, como puede ser un método o los encabezados de nuestra petición, etc.
- Si no especificamos ningún `options`, se ejecutará una simple petición GET, la cual descargará el contenido de lo especificado en el url. El navegador lanzará la petición de inmediato y devolverá una promesa (promise) que luego será utilizada por el código invocado para obtener el resultado. Por lo general, obtener una respuesta es un proceso de dos pasos.

Primero, la promesa promise, devuelta por fetch, resuelve la respuesta con un objeto de la clase incorporada Response tan pronto como el servidor responde con los encabezados de la petición.

En este paso, podemos chequear el status HTTP para poder ver si nuestra petición ha sido exitosa o no, y chequear los encabezados, pero aún no disponemos del cuerpo de la misma.

La promesa es rechazada si el fetch no ha podido establecer la petición HTTP, por ejemplo, por problemas de red o si el sitio especificado en la petición no existe. Estados HTTP anormales, como el 404 o 500 no generan errores. Podemos visualizar los estados HTTP en las propiedades de la respuesta:

- status – código de estado HTTP, por ejemplo: 200.
- ok – booleana, true si el código de estado HTTP es 200 a 299.

Ejemplo:

```
1 let response = await fetch(url);
2
3 if (response.ok) { // si el HTTP-status es 200-299
4   // obtener cuerpo de la respuesta (método debajo)
5   let json = await response.json();
6 } else {
7   alert("Error-HTTP: " + response.status);
8 }
```

Segundo, para obtener el cuerpo de la respuesta, necesitamos utilizar un método adicional. Response provee múltiples métodos basados en promesas para acceder al cuerpo de la respuesta en distintos formatos:

- response.text() – lee y devuelve la respuesta en formato texto,
- response.json() – convierte la respuesta como un JSON,
- response.formData() – devuelve la respuesta como un objeto FormData (explicado en el siguiente capítulo),
- response.blob() – devuelve la respuesta como Blob (datos binarios tipados),
- response.arrayBuffer() – devuelve la respuesta como un objeto ArrayBuffer (representación binaria de datos de bajo nivel),
- Adicionalmente, response.body es un objeto ReadableStream, el cual nos permite acceder al cuerpo como si fuera un stream y leerlo por partes. Veremos un ejemplo de esto más adelante.

Por ejemplo, si obtenemos un objeto de tipo JSON con los últimos commits de GitHub:

```
1 let url = 'https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits';
2 let response = await fetch(url);
3
4 let commits = await response.json(); // leer respuesta del cuerpo y devolver como JSON
5
6 alert(commits[0].author.login);
```

O también usando promesas, en lugar de await:

```
1 fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')
2   .then(response => response.json())
3   .then(commits => alert(commits[0].author.login));
```

Para obtener la respuesta como texto, await response.text() en lugar de .json():

```
1 let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits');
2
3 let text = await response.text(); // leer cuerpo de la respuesta como texto
4
5 alert(text.slice(0, 80) + '...');
```

Como demostración de una lectura en formato binario, hagamos un fetch y mostremos una imagen del logotipo de “*especificación fetch*”:

```
1 let response = await fetch('/article/fetch/logo-fetch.svg');
2
3 let blob = await response.blob(); // download as Blob object
4
5 // crear tag <img> para imagen
6 let img = document.createElement('img');
7 img.style = 'position:fixed;top:10px;left:10px;width:100px';
8 document.body.append(img);
9
10 // mostrar
11 img.src = URL.createObjectURL(blob);
12
13 setTimeout(() => { // ocultar luego de tres segundos
14   img.remove();
15   URL.revokeObjectURL(img.src);
16 }, 3000);
```

**Nota: Si ya obtuvimos la respuesta con response.text(), entonces response.json() no funcionará, dado que el contenido del cuerpo ya ha sido procesado.**

### *Encabezados de respuesta*

Los encabezados de respuesta están disponibles como un objeto de tipo Map dentro del `response.headers`. No es exactamente un Map, pero posee métodos similares para obtener de manera individual encabezados por nombre o si quisiéramos recorrerlos como un objeto:

```
1 let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript-tutorial/branches/main/headings');
2
3 // obtenemos un encabezado
4 alert(response.headers.get('Content-Type')); // application/json; charset=utf-8
5
6 // iteramos todos los encabezados
7 for (let [key, value] of response.headers) {
8   alert(`${key} = ${value}`);
9 }
```

### *Encabezados de petición*

Para especificar un encabezado en nuestro fetch, podemos utilizar la opción `headers`. La misma posee un objeto con los encabezados salientes, como se muestra en el siguiente ejemplo:

```
1 let response = fetch(protectedUrl, {
2   headers: {
3     Authentication: 'secret'
4   }
5 });
```

...Pero existe una lista de encabezados que no pueden ser especificados:

- Accept-Charset, Accept-Encoding, Access-Control-Request-Headers, Access-Control-Request-Method, Connection, Content-Length, Cookie, Cookie2, Date, DNT, Expect, Host, Keep-Alive, Origin, Referer, TE, Trailer, Transfer-Encoding, Upgrade, Via, Proxy-\*, Sec-\*.

Estos encabezados nos aseguran que nuestras peticiones HTTP sean controladas exclusivamente por el navegador, de manera correcta y segura.



## Peticiones POST

Para ejecutar una petición POST, o cualquier otro método, utilizaremos las opciones de fetch:

- method – método HTTP, por ej: POST,
- body – cuerpo de la respuesta, cualquiera de las siguientes:
- cadena de texto (ej. JSON-encoded),
- Objeto FormData, para enviar información como multipart/form-data,
- Blob/BufferSource para enviar información en formato binario,
- URLSearchParams, para enviar información en código x-www-form-urlencoded (no utilizado frecuentemente).

El formato JSON es el más utilizado. Por ejemplo, el código debajo envía la información user como un objeto JSON:

```
1 let user = {
2   nombre: 'Juan',
3   apellido: 'Perez'
4 };
5
6 let response = await fetch('/article/fetch/post/user', {
7   method: 'POST',
8   headers: {
9     'Content-Type': 'application/json;charset=utf-8'
10  },
11   body: JSON.stringify(user)
12 });
13
14 let result = await response.json();
15 alert(result.message);
```

Tener en cuenta, si la respuesta del body es una cadena de texto, entonces el encabezado Content-Type será especificado como text/plain;charset=UTF-8 por defecto.

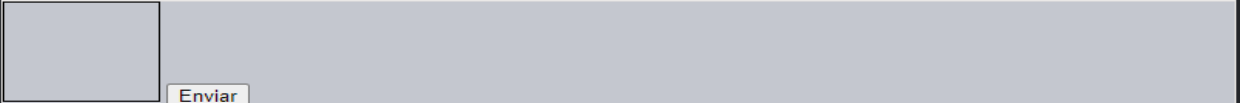
Pero, cómo vamos a enviar un objeto JSON, en su lugar utilizaremos la opción headers especificada a application/json, que es la opción correcta Content-Type para información en formato JSON.

### Enviando una imagen

También es posible enviar datos binarios con fetch, utilizando los objetos Blob o BufferSource.

En el siguiente ejemplo, utilizaremos un <canvas> donde podremos dibujar utilizando nuestro ratón. Haciendo click en el botón “enviar” enviará la imagen al servidor:

```
1 <body style="margin:0">
2   <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>
3
4   <input type="button" value="Enviar" onclick="submit()">
5
6   <script>
7     canvasElem.onmousemove = function(e) {
8       let ctx = canvasElem.getContext('2d');
9       ctx.lineTo(e.clientX, e.clientY);
10      ctx.stroke();
11    };
12
13    async function submit() {
14      let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'))
15      let response = await fetch('/article/fetch/post/image', {
16        method: 'POST',
17        body: blob
18      });
19
20      // el servidor responde con una confirmación y el tamaño de nuestra imagen
21      let result = await response.json();
22      alert(result.message);
23    }
24
25  </script>
26 </body>
```

A screenshot of a web browser window. The browser's address bar is empty. The main content area shows a small, empty square canvas with a thin border. Below the canvas is a button with the text "Enviar".

Una aclaración, aquí no especificamos el Content-Type de manera manual, precisamente porque el objeto Blob posee un tipo incorporado (en este caso image/png, el cual es generado por la función toBlob). Para objetos Blob ese es el valor por defecto del encabezado Content-Type.

Podemos reescribir la función submit() sin utilizar async/await de la siguiente manera:

```
1 function submit() {  
2   canvasElem.toBlob(function(blob) {  
3     fetch('/article/fetch/post/image', {  
4       method: 'POST',  
5       body: blob  
6     })  
7     .then(response => response.json())  
8     .then(result => alert(JSON.stringify(result, null, 2)))  
9   }, 'image/png');  
10 }
```



# RESUMEN

- Una petición fetch típica está formada por dos llamadas await:

```
1 let response = await fetch(url, options); // resuelve con los encabezados de respuesta
2 let result = await response.json(); // accede al cuerpo de respuesta como json
```

- También se puede acceder sin utilizar await:

```
1 fetch(url, options)
2   .then(response => response.json())
3   .then(result => /* procesa resultado */)
```

- Propiedades de respuesta:

- response.status – Código HTTP de la respuesta.
- response.ok – Devuelve true si el código HTTP es 200-299.
- response.headers – Objeto simil-Map que contiene los encabezados HTTP.

- Métodos para obtener el cuerpo de la respuesta:

- response.text() – lee y devuelve la respuesta en formato texto,
- response.json() – convierte la respuesta como un JSON,
- response.formData() – devuelve la respuesta como un objeto FormData (codificación multipart/form-data, explicado en el siguiente capítulo),
- response.blob() – devuelve la respuesta como Blob (datos binarios tipados),
- response.arrayBuffer() – devuelve la respuesta como un objeto ArrayBuffer (datos binarios de bajo nivel)

- Opciones de fetch hasta el momento:

1. method – método HTTP,
2. headers – un objeto los encabezados de la petición (no todos los encabezados están permitidos),
3. body – los datos/información a enviar (cuerpo de la petición) como string, FormData, BufferSource, Blob u objeto UrlSearchParams.

# TAREA

## **Fetch de usuarios de GitHub**

Crear una función async llamada `getUsers(names)`, que tome como parámetro un arreglo de logins de GitHub, obtenga el listado de usuarios de GitHub indicado y devuelva un arreglo de usuarios de GitHub.

La url de GitHub con la información de usuario especifica USERNAME es:

`https://api.github.com/users/USERNAME`.

En el ambiente de prueba (sandbox) hay un ejemplo de referencia.

Detalles a tener en cuenta:

1. Debe realizarse una única petición fetch por cada usuario.
2. Para que la información esté disponible lo antes posible las peticiones no deben ejecutarse de una por vez.
3. Si alguna de las peticiones fallara o si el usuario no existiese, la función debe devolver `null` en el resultado del arreglo.

[Abrir en entorno controlado con pruebas.](#)

# FormData

- Los objetos FormData pueden ser de ayuda en esta tarea. Tal como habrás supuesto, éste es el objeto encargado de representar los datos de los formularios HTML. El constructor es:

```
1 let formData = new FormData([form]);
```

- Si se le brinda un elemento HTML form, el objeto automáticamente capturará sus campos.
- Lo que hace especial al objeto FormData es que los métodos de red, tales como fetch, pueden aceptar un objeto FormData como el cuerpo. Es codificado y enviado como Content-Type: multipart/form-data. Desde el punto de vista del servidor, se ve como una entrega normal.

### Enviando un formulario simple

Enviemos un formulario simple. Tal como se puede ver, es prácticamente una línea:

```
1 <form id="formElem">
2   <input type="text" name="name" value="John">
3   <input type="text" name="surname" value="Smith">
4   <input type="submit">
5 </form>
6
7 <script>
8   formElem.onsubmit = async (e) => {
9     e.preventDefault();
10
11     let response = await fetch('/article/formdata/post/user', {
12       method: 'POST',
13       body: new FormData(formElem)
14     });
15
16     let result = await response.json();
17
18     alert(result.message);
19   };
20 </script>
```

John Smith Enviar

En este ejemplo, el código del servidor no es representado ya que está fuera de nuestro alcance. El servidor acepta la solicitud POST y responde “Usuario registrado”.

## ***Métodos de FormData***

Contamos con métodos para poder modificar los campos del FormData:

- `formData.append(name, value)` – agrega un campo al formulario con el nombre `name` y el valor `value`,
- `formData.append(name, blob, fileName)` – agrega un campo tal como si se tratara de un `<input type="file">`, el tercer argumento `fileName` establece el nombre del archivo (no el nombre del campo), tal como si se tratara del nombre del archivo en el sistema de archivos del usuario,
- `formData.delete(name)` – elimina el campo de nombre `name`,
- `formData.get(name)` – obtiene el valor del campo con el nombre `name`,
- `formData.has(name)` – en caso de que exista el campo con el nombre `name`, devuelve `true`, de lo contrario `false`

Un formulario técnicamente tiene permitido contar con muchos campos con el mismo atributo `name`, por lo que múltiples llamadas a `append` agregarán más campos con el mismo nombre.



Por otra parte existe un método `set`, con la misma sintaxis que `append`. La diferencia está en que `.set` remueve todos los campos con el `name` que se le ha pasado, y luego agrega el nuevo campo. De este modo nos aseguramos de que exista solamente un campo con determinado `name`, el resto es tal como en `append`:

- `formData.set(name, value),`
- `formData.set(name, blob, fileName).`

También es posible iterar por los campos del objeto `formData` utilizando un bucle `for..of`:

```
1 let formData = new FormData();
2 formData.append('key1', 'value1');
3 formData.append('key2', 'value2');
4
5 // Se listan los pares clave/valor
6 for(let [name, value] of formData) {
7   alert(`${name} = ${value}`); // key1 = value1, luego key2 = value2
8 }
```

### Enviando un formulario con un archivo

El formulario siempre es enviado como Content-Type: multipart/form-data, esta codificación permite enviar archivos. Por lo tanto los campos `<input type="file">` también son enviados, tal como sucede en un envío normal.

Aquí un ejemplo con un formulario de este tipo:

```
1 <form id="formElem">
2   <input type="text" name="firstName" value="John">
3   Imagen: <input type="file" name="picture" accept="image/*">
4   <input type="submit">
5 </form>
6
7 <script>
8   formElem.onsubmit = async (e) => {
9     e.preventDefault();
10
11     let response = await fetch('/article/formdata/post/user-avatar', {
12       method: 'POST',
13       body: new FormData(formElem)
14     });
15
16     let result = await response.json();
17
18     alert(result.message);
19   };
20 </script>
```

Imagen:

Ninguno archivo selec.

### Enviando un formulario con datos Blob

Tal como pudimos ver en el capítulo Fetch, es fácil enviar datos binarios generados dinámicamente (por ejemplo una imagen) como Blob. Podemos proporcionarlos directamente en un fetch con el parámetro body.

De todos modos, en la práctica suele ser conveniente enviar la imagen como parte del formulario junto a otra metadata tal como el nombre y no de forma separada. Además los servidores suelen ser más propensos a aceptar formularios multipart, en lugar de datos binarios sin procesar. Este ejemplo envía una imagen desde un <canvas> junto con algunos campos más, como un formulario utilizando FormData:

```
1 <body style="margin:0">
2   <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>
3
4   <input type="button" value="Submit" onclick="submit()">
5
6   <script>
7     canvasElem.onmousemove = function(e) {
8       let ctx = canvasElem.getContext('2d');
9       ctx.lineTo(e.clientX, e.clientY);
10      ctx.stroke();
11    };
12
13    async function submit() {
14      let imageBlob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/
15
16      let formData = new FormData();
17      formData.append("firstName", "John");
18      formData.append("image", imageBlob, "image.png");
19
20      let response = await fetch('/article/formdata/post/image-form', {
21        method: 'POST',
22        body: formData
23      });
24      let result = await response.json();
25      alert(result.message);
26    }
27  </script>
28 </body>
```



Nota como la imagen Blob es agregada:

```
1 formData.append("image", imageBlob, "image.png");
```

Es lo mismo que si hubiera un campo `<input type="file" name="image">` en el formulario, y el usuario enviara un archivo con nombre "image.png" (3er argumento) con los datos imageBlob (2do argumento) desde su sistema de archivos.

El servidor lee el formulario form-data y el archivo tal como si de un formulario regular se tratara.

# RESUMEN

- Los objetos FormData son utilizados para capturar un formulario HTML y enviarlo utilizando fetch u otro método de red. Podemos crear el objeto con `new FormData(form)` desde un formulario HTML, o crear un objeto sin un formulario en absoluto y agregar los campos con los siguientes métodos:
  - `formData.append(nombre, valor)`
  - `formData.append(nombre, blob, nombreDeArchivo)`
  - `formData.set(nombre, valor)`
  - `formData.set(nombre, blob, nombreDeArchivo)`
- Nótese aquí dos particularidades:
  1. El método `set` remueve campos con el mismo nombre, mientras que `append` no. Esta es la única diferencia entre estos dos métodos.
  2. Para enviar un archivo, se requiere de tres argumentos, el último argumento es el nombre del archivo, el cual normalmente es tomado desde el sistema de archivos del usuario por el `<input type="file">`.
- Otros métodos son:
  - `formData.delete(nombre)`
  - `formData.get(nombre)`
  - `formData.has(nombre)`