DWC

(Desarrollo Web en entorno cliente)

Índice

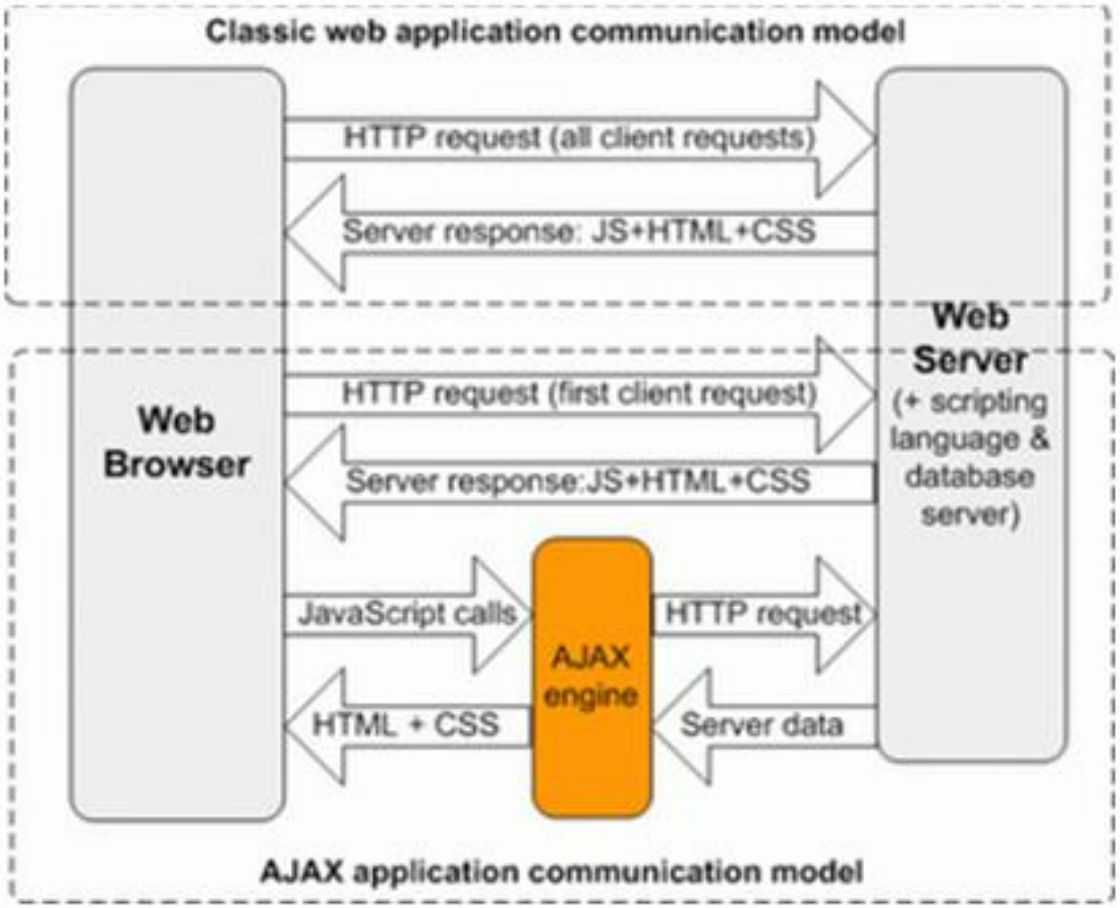
1.- AJAX 1
2.- XMLHttpRequest (XHR) 2
2.1- Funcionamiento del objeto XMLHttpRequest 2
2.2.- Estados de una petición 6
2.3.- Solicitud de cancelación de una petición 7
2.4.- Parámetros de búsqueda de URL 7
2.5.- Peticiones síncronas 8
2.6.- Encabezados HTTP 9
2.7.- Enviar datos al servidor POST, PUT y PATCH 10
2.7.1.- Mandar datos con un formulario 10
2.7.2.- Enviar datos en formato JSON 11
2.8.- Subir archivos 12
2.9.- Peticiones cross-origin 14
2.10.- Resumen 14

1.- 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



2.- XMLHttpRequest (XHR)

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 objeto XMLHttpRequest tiene dos modos de operación: **síncrono** y **asíncrono**.

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, generalmente justo después de new XMLHttpRequest:**

```
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 sincróna
- **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 estara "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() { // if the request couldn't be made at all
  alert(`Network Error`);
};
```

- **progress** se activa periódicamente mientras se descarga la respuesta, informa cuánto se ha descargado.

```
xhr.onprogress = function(event) { // triggers periodically
  // event.loaded - how many bytes downloaded
  // event.lengthComputable = true if the server sent Content-Length header
  // event.total - total number of bytes (if lengthComputable)
  alert(`Received ${event.loaded} of ${event.total}`);
};
```

El siguiente código muestra el proceso de descarga de datos de la petición a la url /article/xmlhttprequest/example/load

```
// 1. Create a new XMLHttpRequest object
let xhr = new XMLHttpRequest();

// 2. Configure it: GET-request for the URL /article/.../load
xhr.open('GET', '/article/xmlhttprequest/example/load');

// 3. Send the request over the network
xhr.send();

// 4. This will be called after the response is received
xhr.onload = function() {
  if (xhr.status != 200) { // analyze HTTP status of the response
    alert(`Error ${xhr.status}: ${xhr.statusText}`); // e.g. 404: Not Found
  } else { // show the result
    alert(`Done, got ${xhr.response.length} bytes`); // response is the server
  }
};
```

```
xhr.onprogress = function(event) {
  if (event.lengthComputable) {
    alert(`Received ${event.loaded} of ${event.total} bytes`);
  } else {
    alert(`Received ${event.loaded} bytes`); // no Content-Length
  }

};

xhr.onerror = function() {
  alert("Request failed");
};
```

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, 404, 403y así sucesivamente
   - **statusText**

     Mensaje de estado HTTP (una cadena): generalmente OK para 200, Not Found para 404, Forbidden para, 403 etc.
   - **response** (en scripts antiguos responseText)

     La respuesta del servidor.

   Tipos de respuestas

   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).

Por ejemplo, para obtener la respuesta como JSON:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/example/json');

xhr.responseType = 'json';

xhr.send();

// the response is {"message": "Hello, world!"}
xhr.onload = function() {
  let responseObj = xhr.response;
  alert(responseObj.message); // Hello, world!
};
```

En los scripts antiguos también puede encontrar las propiedades **xhr.responseText** e incluso **xhr.responseXML**.
Existen por razones históricas, para obtener una cadena o un documento XML. Hoy en día, debemos establecer el formato **xhr.responseType** y obtener **xhr.response** como se demostró anteriormente.

2.2.- Estados de una petición

XMLHttp Request cambia entre estados a medida que avanza. El estado actual es accesible como **xhr.readyState**.

Los estados de una petición son:

- **UNSENT = 0**; // initial state
- **OPENED = 1**; // open called
- **HEADERS_RECEIVED = 2**; // response headers received

- **LOADING = 3**; // response is loading (a data packed is received)
- **DONE = 4**; // request complete

Un objeto XMLHttpRequest los recorre en el orden 0→ 1→ 2→ 3→… → 3→ 4. El estado se 3 repite cada vez que se recibe un paquete de datos a través de la red.

Podemos rastrearlos usando el evento **readystatechange**:

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 3) {
    // loading
  }
  if (xhr.readyState == 4) {
    // request finished
  }
};
```

Se puede encontrar readystatechange en código antiguo, está allí por razones históricas, ya que **hubo un momento en que no hubo load y otros eventos**. Hoy en día, los manejadores load/error/progress lo desprecian.

2.3.- Solicitud de cancelación de una petición

Podemos finalizar la solicitud en cualquier momento mediante la llamada a **xhr.abort()**

```
xhr.abort(); // terminate the request
```

Eso desencadena el evento **abort**, y convierte xhr.status se a0.

2.4.- Parámetros de búsqueda de URL

Para agregar parámetros a la URL, como ?name=valuey y asegurar la codificación adecuada, podemos usar el objeto URL :

```
let url = new URL('https://google.com/search');
url.searchParams.set('q', 'test me!');

// the parameter 'q' is encoded
xhr.open('GET', url); // https://google.com/search?q=test+me%21
```

O directamente introducir nosotros los parámetros en formato queryString en la url

```
// the parameter 'q' is in the url
let url=’ https://google.com/search?q=’+'test me!'
xhr.open('GET', url)
```

2.5.- Peticiones síncronas

Si en el método open el tercer parámetro **async** está establecido en **false**, la solicitud se realiza de forma sincrónica.
En otras palabras, la ejecución de JavaScript se detiene con send() y se reanuda cuando se recibe la respuesta. Algo parecido a los comandos **alert** o **prompt**.

Las llamadas síncronas se usan raramente, porque bloquean JavaScript en la página hasta que se completa la carga. En algunos navegadores se hace imposible desplazarse. Si una llamada síncrona toma demasiado tiempo, el navegador puede sugerir cerrar la página web "suspendida".

Muchas capacidades avanzadas de XMLHttpRequest, como solicitar desde otro dominio o especificar un tiempo de espera, no están disponibles para solicitudes síncronas. Ademásno hay indicación de progreso.

Debido a todo eso, las solicitudes síncronas no se utilizan, casi nunca.

2.6.- Encabezados HTTP

XMLHttpRequest permite enviar encabezados personalizados y leer encabezados de la respuesta.

Hay 3 métodos para los encabezados HTTP:

- **setRequestHeader(name, value)**

  Establece el encabezado de solicitud con el dado name y value.

```
xhr.setRequestHeader('Content-Type', 'application/json');
```

  Este encabezado lo utilizaremos siempre para mandar información en formato JSON y que el servidor sepa que le va a llegar un JSON.

  Una vez que se establece el encabezado, se establece. Las llamadas adicionales agregan información al encabezado, no lo sobrescribe.

  Por ejemplo:

```
xhr.setRequestHeader('X-Auth', '123');
xhr.setRequestHeader('X-Auth', '456');

// the header will be:
// X-Auth: 123, 456
```

- **getResponseHeader(name)**

  Obtiene el encabezado de respuesta con el name dado

  Por ejemplo:

```
xhr.getResponseHeader('Content-Type')
```

- **getAllResponseHeaders()**

  Devuelve todos los encabezados de respuesta. Los encabezados se devuelven como una sola línea, por ejemplo:

```
Cache-Control: max-age=31536000
Content-Length: 4260
Content-Type: image/png
```

```
Date: Sat, 08 Sep 2020 16:53:16 GMT
```

2.7.- Enviar datos al servidor POST, PUT y PATCH

Para hacer una solicitud POST o cualquier otra en la que debamos mandar datos al servidor, podemos hacerlo de dos formas:

1. usar un formulario con el objeto FormData incorporado
2. mandar un objeto JSON con los datos de un formulario

2.7.1.- Mandar datos con un formulario

Creamos un objeto de tipo Form, y opcionalmente rellenamos su contenido con un formulario existente. Con **append** podemos añadir más campos si es necesario:

```
let formData = new FormData([form]); // creates an object, optionally fill from <form>

formData.append(name, value); // appends a field
```

Una vez configurada la información a mandar usamos el método open con POST

```
xhr.open('POST', ...)-
```

Finalmente lanzamos la petición con el método send con los datos en su interior

```
xhr.send(formData) para enviar el formulario al servidor.
```

Por ejemplo:

```
<form name="person">
  <input name="name" value="John">
  <input name="surname" value="Smith">
</form>

<script>
  // pre-fill FormData from the form
  let formData = new FormData(document.forms.person);

  // add one more field
```

```
  formData.append("middle", "Lee");

  // send it out
  let xhr = new XMLHttpRequest();
  xhr.open("POST", "/article/xmlhttprequest/post/user");
  xhr.send(formData);

  xhr.onload = () => alert(xhr.response);
</script>
```

El formulario se envía con una codificación **multipart/form-data.**

2.7.2.- Enviar datos en formato JSON

Muchas veces los datos a mandar al servidor no tienen porque venir de un formulario, puede que ya tengamos un objeto y que queramos enviar ese objeto al servidor.

En estos casos 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.stringifyy(objeto)**. Muchos servidores decodifican directamente de JSON, con lo cual a veces no es necesario incluirlo.

```
let xhr = new XMLHttpRequest();

let json = JSON.stringify({ name: "John",surname: "Smith"});

xhr.open("POST", '/submit')
xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');

xhr.send(json);
```

La mayoría de veces la forma de subir datos al servidor es a través de un objeto JSON evitando tener que utilizar el objeto formData. Incluso cuando los datos se han de mandar de un formulario en el que el usuario ha introducido los valores adecuados se suele utilizar un objeto JSON, para ello lo que se hace es serializar el formulario a un objeto JSON. Esto consiste en crear un objeto JSON con los pares **key:value** igual al **name** de los campos y al **valor** de cada uno de ellos

2.8.- Subir archivos

El evento **onprogress** se dispara solo en la etapa de descarga, es decir cuando estamos esperando recibir datos desde el servidor.

También tenemos el otro caso cuando hacemos un POST de algo grande al servidor. En este caso XMLHttpRequest primero carga nuestros datos (el cuerpo de la solicitud) y luego descarga la respuesta.

Si estamos subiendo algo grande, seguramente estaremos muy interesados en seguir el progreso de la carga. Pero xhr.onprogress no interviene en esta situación.

Hay otro objeto, sin métodos, exclusivamente dedicado al seguimiento de los eventos de carga: **xhr.upload**.

Genera eventos similares xhr, pero los xhr.upload se activan únicamente al cargar (subir archivos):

- **loadstart** comenzó la carga.
- **progress** se activa periódicamente durante la carga.
- **abort** carga abortada.
- **error** error no HTTP.
- **load c**arga finalizada con éxito.
- **timeout**- tiempo de espera de carga (si timeoutse establece la propiedad).
- l**oadend** - la carga finalizó con éxito o error.

Ejemplo de manejadores:

```
xhr.upload.onprogress = function(event) {
  alert(`Uploaded ${event.loaded} of ${event.total} bytes`);
};
```

```
xhr.upload.onload = function() {
  alert(`Upload finished successfully.`);
};
```

```
xhr.upload.onerror = function() {
  alert(`Error during the upload: ${xhr.status}`);
};
```

Aquí hay un ejemplo de la vida real: carga de archivos con indicación de progreso:

```
<input type="file" onchange="upload(this.files[0])">

<script>
function upload(file) {
  let xhr = new XMLHttpRequest();

  // track upload progress
  xhr.upload.onprogress = function(event) {
    console.log(`Uploaded ${event.loaded} of ${event.total}`);
  };

  // track completion: both successful or not
  xhr.onloadend = function() {
    if (xhr.status == 200) {
      console.log("success");
    } else {
      console.log("error " + this.status);
    }
```

```
  };

  xhr.open("POST", "/article/xmlhttprequest/post/upload");
  xhr.send(file);
}
</script>
```

2.9.- Peticiones cross-origin

XMLHttpRequest puede realizar solicitudes de origen cruzado utilizando la misma política CORS.

XMLHttpRequest no envía cookies y autorizaciones HTTP a otro origen por defecto. Para habilitarlos, hay que configurar **xhr.withCredentials** a **true**

```
let xhr = new XMLHttpRequest();
xhr.withCredentials = true;

xhr.open('POST', 'http://anywhere.com/request');
```

Consulte el capítulo Recuperar: solicitudes de origen cruzado para obtener detalles sobre los encabezados de origen cruzado.

2.10.- Resumen

Código típico de la solicitud GET con XMLHttpRequest:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/my/url');

xhr.send();

xhr.onload = function() {
  if (xhr.status != 200) { // HTTP error?
    // handle error
```

```
    alert( 'Error: ' + xhr.status);
    return;
  }

  // get the response from xhr.response
};

xhr.onprogress = function(event) {
  // report progress
  alert(`Loaded ${event.loaded} of ${event.total}`);
};

xhr.onerror = function() {
  // handle non-HTTP error (e.g. network down)
};
```

En realidad, hay más eventos, la especificación moderna los enumera (en el orden del ciclo de vida):

- **loadstart** - la solicitud ha comenzado.
- **progress-** ha llegado un paquete de datos de la respuesta, todo el cuerpo de la respuesta está en este momento response.
- **abort**- la llamada canceló la solicitud xhr.abort().
- **error**- se ha producido un error de conexión, por ejemplo, un nombre de dominio incorrecto. No sucede con errores HTTP como 404.
- **load** - la solicitud ha finalizado con éxito.
- **timeout** - la solicitud se canceló debido al tiempo de espera (solo ocurre si se configuró).
- **loadend**- desencadenantes después load, error, timeouto abort.

Los **error**, **abort**, **timeou**t, y **loadeventos** son mutuamente excluyentes. **Solo uno de ellos puede suceder**.

Los eventos más utilizados son la finalización de carga (**load**), el fallo de carga (**error**), o podemos usar un solo controlador **loadend** y verificar las propiedades del objeto de solicitud **xhr** para ver qué sucedió.

Hemos visto otro evento: **readystatechange**. Históricamente, apareció hace mucho tiempo, antes de que se estableciera la especificación actual. Hoy en día, no hay necesidad de usarlo, podemos reemplazarlo con eventos más nuevos, pero a menudo se puede encontrar en scripts más antiguos.

Si necesitamos comprobar la carga específicamente, entonces deberíamos escuchar los mismos eventos en el **xhr.uploadobjeto**.