24 de octubre de 2022

XMLHttpRequest

`XMLHttpRequest` es un objeto nativo del navegador que permite hacer solicitudes HTTP desde JavaScript.

A pesar de tener la palabra "XML" en su nombre, se puede operar sobre cualquier dato, no solo en formato XML. Podemos cargar y descargar archivos, dar seguimiento y mucho más.

Ahora hay un método más moderno `fetch` que en algún sentido hace obsoleto a `XMLHttpRequest` .

En el desarrollo web moderno `XMLHttpRequest` se usa por tres razones:

1. Razones históricas: necesitamos soportar scripts existentes con `XMLHttpRequest` .
2. Necesitamos soportar navegadores viejos, y no queremos `polyfills` (p.ej. para mantener los scripts pequeños).
3. Necesitamos hacer algo que `fetch` no puede todavía, ej. rastrear el progreso de subida.

¿Te suena familiar? Si es así, está bien, adelante con `XMLHttpRequest` . De otra forma, por favor, dirígete a Fetch.

Lo básico

XMLHttpRequest tiene dos modos de operación: sincrónica y asíncrona.

Veamos primero la asíncrona, ya que es utilizada en la mayoría de los casos.

Para hacer la petición, necesitamos seguir 3 pasos:

1. Crear el objeto `XMLHttpRequest` :

```
let xhr = new XMLHttpRequest();
```

El constructor no tiene argumentos.

2. Inicializarlo, usualmente justo después de `new XMLHttpRequest` :

```
xhr.open(method, URL, [async, user, password])
```

Este método especifica los parámetros principales para la petición:

- `method` – método HTTP. Usualmente `"GET"` o `"POST"` .
- `URL` – la URL a solicitar, una cadena, puede ser un objeto URL.
- `async` – si se asigna explícitamente a `false` , entonces la petición será asincrónica. Cubriremos esto un poco más adelante.
- `user` , `password` – usuario y contraseña para autenticación HTTP básica (si se requiere).

Por favor, toma en cuenta que la llamada a `open` , contrario a su nombre, no abre la conexión. Solo configura la solicitud, pero la actividad de red solo empieza con la llamada del método `send` .

3. Enviar.

```
xhr.send([body])
```

Este método abre la conexión y envía ka solicitud al servidor. El parámetro adicional `body` contiene el cuerpo de la solicitud.

Algunos métodos como `GET` no tienen un cuerpo. Y otros como `POST` usan el parámetro `body` para enviar datos al servidor. Vamos a ver unos ejemplos de eso más tarde.

4. Escuchar los eventos de respuesta `xhr` .

Estos son los tres eventos más comúnmente utilizados:

- `load` – cuando la solicitud está; completa (incluso si el estado HTTP es 400 o 500), y la respuesta se descargó por completo.
- `error` – cuando la solicitud no pudo ser realizada satisfactoriamente, ej. red caída o una URL inválida.
- `progress` – se dispara periódicamente mientras la respuesta está siendo descargada, reporta cuánto se ha descargado.

```
xhr.onload = function() {
  alert(`Cargado: ${xhr.status} ${xhr.response}`);
};

xhr.onerror = function() { // solo se activa si la solicitud no se puede realizar
  alert(`Error de red`);
};

xhr.onprogress = function(event) { // se dispara periódicamente
  // event.loaded - cuántos bytes se han descargado
  // event.lengthComputable = devuelve true si el servidor envía la cabecera Content-Leng
  // event.total - número total de bytes (si `lengthComputable` es `true`)
  alert(`Recibido ${event.loaded} of ${event.total}`);
};
```

Aquí un ejemplo completo. El siguiente código carga la URL en `/article/xmlhttprequest/example/load` desde el servidor e imprime el progreso:

```
// 1. Crea un nuevo objeto XMLHttpRequest
let xhr = new XMLHttpRequest();

// 2. Configuración: solicitud GET para la URL /article/.../load
xhr.open('GET', '/article/xmlhttprequest/example/load');

// 3. Envía la solicitud a la red
xhr.send();
```

```
9
10  // 4. Esto se llamará después de que la respuesta se reciba
11  xhr.onload = function() {
12    if (xhr.status != 200) { // analiza el estado HTTP de la respuesta
13      alert(`Error ${xhr.status}: ${xhr.statusText}`); // ej. 404: No encontrado
14    } else { // muestra el resultado
15      alert(`Hecho, obtenidos ${xhr.response.length} bytes`); // Respuesta del servidor
16    }
17  };
18
19  xhr.onprogress = function(event) {
20    if (event.lengthComputable) {
21      alert(`Recibidos ${event.loaded} de ${event.total} bytes`);
22    } else {
23      alert(`Recibidos ${event.loaded} bytes`); // sin Content-Length
24    }
25
26  };
27
28  xhr.onerror = function() {
29    alert("Solicitud fallida");
30  };
```

Una vez el servidor haya respondido, podemos recibir el resultado en las siguientes propiedades de `xhr`:

`status`

Código del estado HTTP (un número): `200`, `404`, `403` y así por el estilo, puede ser `0` en caso de una falla no HTTP.

`statusText`

Mensaje del estado HTTP (una cadena): usualmente `OK` para `200`, `Not Found` para `404`, `Forbidden` para `403` y así por el estilo.

`response` (scripts antiguos deben usar `responseText`)

El cuerpo de la respuesta del servidor.

También podemos especificar un tiempo límite usando la propiedad correspondiente:

```
xhr.timeout = 10000; // límite de tiempo en milisegundos, 10 segundos
```

Si la solicitud no es realizada con éxito dentro del tiempo dado, se cancela y el evento `timeout` se activa.

Parámetros de búsqueda URL

Para agregar los parámetros a la URL, como `?nombre=valor`, y asegurar la codificación adecuada, podemos utilizar un objeto URL:

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

// el parámetro 'q' está codificado
xhr.open('GET', url); // https://google.com/search?q=test+me%21
```

Tipo de respuesta

Podemos usar la propiedad `xhr.responseType` para asignar el formato de la respuesta:

- `""` (default) – obtiene una cadena,
- `"text"` – obtiene una cadena,
- `"arraybuffer"` – obtiene un `ArrayBuffer` (para datos binarios, ve el capítulo ArrayBuffer, arrays binarios),
- `"blob"` – obtiene un `Blob` (para datos binarios, ver el capítulo Blob),
- `"document"` – obtiene un documento XML (puede usar XPath y otros métodos XML) o un documento HTML (en base al tipo MIME del dato recibido),

- `"json"` – obtiene un JSON (automáticamente analizado).

Por ejemplo, obtengamos una respuesta como JSON:

```
let xhr = new XMLHttpRequest();

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

xhr.responseType = 'json';

xhr.send();

// la respuesta es {"message": "Hola, Mundo!"}
xhr.onload = function() {
  let responseObj = xhr.response;
  alert(responseObj.message); // Hola, Mundo!
};
```

Por favor tome nota:

En los scripts antiguos puedes encontrar también las propiedades `xhr.responseText` e incluso `xhr.responseXML` .

Existen por razones históricas, para obtener ya sea una cadena o un documento XML. Hoy en día, debemos seleccionar el formato en `xhr.responseType` y obtener `xhr.response` como se demuestra debajo.

Estados

`XMLHttpRequest` cambia entre estados a medida que avanza. El estado actual es accesible como `xhr.readyState` .

Todos los estados, como en la especificación:

```
UNSENT = 0; // estado inicial
OPENED = 1; // llamada abierta
HEADERS_RECEIVED = 2; // cabeceras de respuesta recibidas
LOADING = 3; // la respuesta está cargando (un paquete de datos es recibido)
DONE = 4; // solicitud completa
```

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

Podemos seguirlos usando el evento `readystatechange` :

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 3) {
    // cargando
  }
  if (xhr.readyState == 4) {
    // solicitud finalizada
  }
};
```

Puedes encontrar oyentes del evento `readystatechange` en código realmente viejo, está ahí por razones históricas, había un tiempo cuando no existían `load` y otros eventos. Hoy en día los manipuladores `load/error/progress` lo hacen obsoleto.

Abortando solicitudes

Podemos terminar la solicitud en cualquier momento. La llamada a `xhr.abort()` hace eso:

```
xhr.abort(); // termina la solicitud
```

Este dispara el evento `abort` , y el `xhr.status` se convierte en `0` .

Solicitudes sincrónicas

Si en el método `open` el tercer parámetro `async` se asigna como `false`, la solicitud se hace sincrónicamente.

En otras palabras, la ejecución de JavaScript se pausa en el `send()` y se reanuda cuando la respuesta es recibida. Algo como los comandos `alert` o `prompt`.

Aquí está el ejemplo reescrito, el tercer parámetro de `open` es `false`:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/hello.txt', false);

try {
  xhr.send();
  if (xhr.status != 200) {
    alert(`Error ${xhr.status}: ${xhr.statusText}`);
  } else {
    alert(xhr.response);
  }
} catch(err) { // en lugar de onerror
  alert("Solicitud fallida");
}
```

Puede verse bien, pero las llamadas sincrónicas son rara vez utilizadas porque bloquean todo el JavaScript de la página hasta que la carga está completa. En algunos navegadores se hace imposible hacer scroll. Si una llamada síncrona toma mucho tiempo, el navegador puede sugerir cerrar el sitio web "colgado".

Algunas capacidades avanzadas de `XMLHttpRequest`, como solicitar desde otro dominio o especificar un tiempo límite, no están disponibles para solicitudes síncronas. Tampoco, como puedes ver, la indicación de progreso.

La razón de esto es que las solicitudes sincrónicas son utilizadas muy escasamente, casi nunca. No hablaremos más sobre ellas.

Cabeceras HTTP

`XMLHttpRequest` permite tanto enviar cabeceras personalizadas como leer cabeceras de la respuesta.

Existen 3 métodos para las cabeceras HTTP:

`setRequestHeader(name, value)`

Asigna la cabecera de la solicitud con los valores `name` y `value` provistos.

Por ejemplo:

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

Limitaciones de cabeceras

Muchas cabeceras se administran exclusivamente por el navegador, ej. `Referer` y `Host` . La lista completa está en la especificación.

`XMLHttpRequest` no está permitido cambiarlos, por motivos de seguridad del usuario y la exactitud de la solicitud.

No se pueden eliminar cabeceras

Otra peculiaridad de `XMLHttpRequest` es que no puede deshacer un `setRequestHeader`.

Una vez que una cabecera es asignada, ya está asignada. Llamadas adicionales agregan información a la cabecera, no la sobreescriben.

Por ejemplo:

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

// la cabecera será:
// X-Auth: 123, 456
```

`getResponseHeader(name)`

Obtiene la cabecera de la respuesta con el `name` dado (excepto `Set-Cookie` y `Set-Cookie2`).

Por ejemplo:

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

`getAllResponseHeaders()`

Devuelve todas las cabeceras de la respuesta, excepto por `Set-Cookie` y `Set-Cookie2`.

Las cabeceras se devuelven como una sola línea, ej.:

```
Cache-Control: max-age=31536000
Content-Length: 4260

```

```
4  Content-Type: image/png
   Date: Sat, 08 Sep 2012 16:53:16 GMT
```

El salto de línea entre las cabeceras siempre es un `"\r\n"` (independiente del SO), así podemos dividirlas en cabeceras individuales. El separador entre el nombre y el valor siempre es dos puntos seguido de un espacio `": "`. Eso quedó establecido en la especificación.

Así, si queremos obtener un objeto con pares nombre/valor, necesitamos tratarlas con un poco de JS.

Como esto (asumiendo que si dos cabeceras tienen el mismo nombre, entonces el último sobreescribe al primero):

```
1  let headers = xhr
2    .getAllResponseHeaders()
3    .split('\r\n')
4    .reduce((result, current) => {
5      let [name, value] = current.split(': ');
6      result[name] = value;
7      return result;
8    }, {});
9
10 // headers['Content-Type'] = 'image/png'
```

POST, Formularios

Para hacer una solicitud POST, podemos utilizar el objeto FormData nativo.

La sintaxis:

```
1  let formData = new FormData([form]); // crea un objeto, opcionalmente se completa con un <f
2  formData.append(name, value); // añade un campo
```

Lo creamos, opcionalmente lleno desde un formulario, `append` (agrega) más campos si se necesitan, y entonces:

1. `xhr.open('POST', ...)` – se utiliza el método `POST` .
2. `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 llenado del objeto FormData desde el formulario
  let formData = new FormData(document.forms.person);

  // agrega un campo más
  formData.append("middle", "Lee");

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

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

El formulario fue enviado con codificación `multipart/form-data` .

O, si nos gusta más JSON, entonces, un `JSON.stringify` y lo enviamos como un string.

Solo no te olvides de asignar la cabecera `Content-Type: application/json` , muchos frameworks del lado del servidor decodifican automáticamente JSON con este:

```
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);
```

El método `.send(body)` es bastante omnívoro. Puede enviar casi cualquier `body`, incluyendo objetos `Blob` y `BufferSource`.

Progreso de carga

El evento `progress` se dispara solo en la fase de descarga.

Esto es: si hacemos un `POST` de algo, `XMLHttpRequest` primero sube nuestros datos (el cuerpo de la respuesta), entonces descarga la respuesta.

Si estamos subiendo algo grande, entonces seguramente estaremos interesados en rastrear el progreso de nuestra carga. Pero `xhr.onprogress` no ayuda aquí.

Hay otro objeto, sin métodos, exclusivamente para rastrear los eventos de subida: `xhr.upload`.

Este genera eventos similares a `xhr`, pero `xhr.upload` se dispara solo en las subidas:

- `loadstart` – carga iniciada.
- `progress` – se dispara periódicamente durante la subida.
- `abort` – carga abortada.
- `error` – error no HTTP.
- `load` – carga finalizada con éxito.
- `timeout` – carga caducada (si la propiedad `timeout` está asignada).

- `loadend` – carga finalizada con éxito o error.

Ejemplos 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 durante la carga: ${xhr.status}`);
};
```

Aquí un ejemplo de la vida real: indicación del progreso de subida de un archivo:

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

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

  // rastrea el progreso de la subida
  xhr.upload.onprogress = function(event) {
    console.log(`Uploaded ${event.loaded} of ${event.total}`);
  };

  // seguimiento completado: sea satisfactorio o no
  xhr.onloadend = function() {
    if (xhr.status == 200) {
      console.log("Logrado");
```

```
    } else {
      console.log("error " + this.status);
    }
  };

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

Solicitudes de origen cruzado (Cross-origin)

`XMLHttpRequest` puede hacer solicitudes de origen cruzado, utilizando la misma política CORS que se solicita.

Tal como `fetch`, no envía cookies ni autorización HTTP a otro origen por omisión. Para activarlas, asigna `xhr.withCredentials` como `true`:

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

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

Ve el capítulo Fetch: Cross-Origin Requests para detalles sobre las cabeceras de origen cruzado.

Resumen

Codificación típica de la solicitud GET con `XMLHttpRequest`:

```
let xhr = new XMLHttpRequest();

```

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

xhr.send();

xhr.onload = function() {
  if (xhr.status != 200) { // error HTTP?
    // maneja el error
    alert( 'Error: ' + xhr.status);
    return;
  }

  // obtiene la respuesta de xhr.response
};

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

xhr.onerror = function() {
  // manejo de un error no HTTP (ej. red caída)
};
```

De hecho hay más eventos, la especificación moderna los lista (en el orden del ciclo de vida):

- `loadstart` – la solicitud ha empezado.
- `progress` – un paquete de datos de la respuesta ha llegado, el cuerpo completo de la respuesta al momento está en `response`.
- `abort` – la solicitud ha sido cancelada por la llamada de `xhr.abort()`.
- `error` – un error de conexión ha ocurrido, ej. nombre de dominio incorrecto. No pasa con errores HTTP como 404.
- `load` – la solicitud se ha completado satisfactoriamente.
- `timeout` – la solicitud fue cancelada debido a que caducó (solo pasa si fue configurado).
- `loadend` – se dispara después de `load`, `error`, `timeout` o `abort`.

Los eventos `error`, `abort`, `timeout`, y `load` son mutuamente exclusivos. Solo uno de ellos puede pasar.

Los eventos más usados son la carga terminada (`load`), falla de carga (`error`), o podemos usar un solo manejador `loadend` y comprobar las propiedades del objeto solicitado `xhr` para ver qué ha pasado.

Ya hemos visto otro evento: `readystatechange` . Históricamente, apareció hace mucho tiempo, antes de que la especificación fuera publicada. Hoy en día no es necesario usarlo; podemos reemplazarlo con eventos más nuevos, pero puede ser encontrado a menudo en scripts viejos.

Si necesitamos rastrear específicamente, entonces debemos escuchar a los mismos eventos en el objeto `xhr.upload` .

Lección anterior Próxima lección

Compartir

Mapa del Tutorial

Comentarios

- Si tiene sugerencias sobre qué mejorar, por favor enviar una propuesta de GitHub o una solicitud de extracción en lugar de comentar.
- Si no puede entender algo en el artículo, por favor explique.
- Para insertar algunas palabras de código, use la etiqueta `<code>` , para varias líneas – envolverlas en la etiqueta `<pre>` , para más de 10 líneas – utilice una entorno controlado (sandbox) (plnkr, jsbin, codepen...)

acerca del proyecto contáctenos