14th August 2019

XMLHttpRequest

`XMLHttpRequest` is a built-in browser object that allows to make HTTP requests in JavaScript.

Despite of having the word "XML" in its name, it can operate on any data, not only in XML format. We can upload/download files, track progress and much more.

Right now, there's another, more modern method `fetch`, that somewhat deprecates `XMLHttpRequest`.

In modern web-development `XMLHttpRequest` is used for three reasons:

1. Historical reasons: we need to support existing scripts with `XMLHttpRequest`.
2. We need to support old browsers, and don't want polyfills (e.g. to keep scripts tiny).
3. We need something that `fetch` can't do yet, e.g. to track upload progress.

Does that sound familiar? If yes, then all right, go on with `XMLHttpRequest`. Otherwise, please head on to Fetch.

The basics

XMLHttpRequest has two modes of operation: synchronous and asynchronous.

Let's see the asynchronous first, as it's used in the majority of cases.

To do the request, we need 3 steps:

1. Create `XMLHttpRequest`:

```
let xhr = new XMLHttpRequest(); // the constructor has no arguments
```

2. Initialize it:

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

This method is usually called right after `new XMLHttpRequest`. It specifies the main parameters of the request:

- `method` – HTTP-method. Usually `"GET"` or `"POST"`.
- `URL` – the URL to request, a string, can be URL object.
- `async` – if explicitly set to `false`, then the request is synchronous, we'll cover that a bit later.
- `user`, `password` – login and password for basic HTTP auth (if required).

Please note that `open` call, contrary to its name, does not open the connection. It only configures the request, but the network activity only starts with the call of `send`.

3. Send it out.

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

This method opens the connection and sends the request to server. The optional `body` parameter contains the request body.

Some request methods like `GET` do not have a body. And some of them like `POST` use `body` to send the data to the server. We'll see examples later.

4. Listen to `xhr` events for response.

These three are the most widely used:

- `load` – when the result is ready, that includes HTTP errors like 404.
- `error` – when the request couldn't be made, e.g. network down or invalid URL.
- `progress` – triggers periodically during the download, reports how much downloaded.

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

xhr.onerror = function() { // only triggers if the request couldn't be made at all
  alert(`Network Error`);
};

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}`);
};
```

Here's a full example. The code below loads the URL at `/article/xmlhttprequest/example/load` from the server and prints the progress:

```
// 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`); // responseText 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");
};
```

Once the server has responded, we can receive the result in the following `xhr` properties:

`status`

HTTP status code (a number): `200`, `404`, `403` and so on, can be `0` in case of a non-HTTP failure.

`statusText`

HTTP status message (a string): usually `OK` for `200`, `Not Found` for `404`, `Forbidden` for `403` and so on.

`response` (old scripts may use `responseText`)

The server response body.

We can also specify a timeout using the corresponding property:

```
xhr.timeout = 10000; // timeout in ms, 10 seconds
```

If the request does not succeed within the given time, it gets canceled and `timeout` event triggers.

URL search parameters

To add parameters to URL, like `?name=value`, and ensure the proper encoding, we can use URL object:

```
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
```

Response Type

We can use `xhr.responseType` property to set the response format:

- `""` (default) – get as string,
- `"text"` – get as string,
- `"arraybuffer"` – get as `ArrayBuffer` (for binary data, see chapter ArrayBuffer, binary arrays),
- `"blob"` – get as `Blob` (for binary data, see chapter),
- `"document"` – get as XML document (can use XPath and other XML methods),
- `"json"` – get as JSON (parsed automatically).

For example, let's get the response as 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!
};
```

Please note:

In the old scripts you may also find `xhr.responseText` and even `xhr.responseXML` properties.

They exist for historical reasons, to get either a string or XML document. Nowadays, we should set the format in `xhr.responseType` and get `xhr.response` as demonstrated above.

Ready states

`XMLHttpRequest` changes between states as it progresses. The current state is accessible as `xhr.readyState`.

All states, as in the specification:

```
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
```

An `XMLHttpRequest` object travels them in the order `0` → `1` → `2` → `3` → ... → `3` → `4`. State `3` repeats every time a data packet is received over the network.

We can track them using `readystatechange` event:

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 3) {
```

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

You can find `readystatechange` listeners in really old code, it's there for historical reasons, as there was a time when there were no `load` and other events. Nowadays, `load/error/progress` handlers deprecate it.

Aborting request

We can terminate the request at any time. The call to `xhr.abort()` does that:

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

That triggers `abort` event, and `xhr.status` becomes `0`.

Synchronous requests

If in the `open` method the third parameter `async` is set to `false`, the request is made synchronously.

In other words, JavaScript execution pauses at `send()` and resumes when the response is received. Somewhat like `alert` or `prompt` commands.

Here's the rewritten example, the 3rd parameter of `open` is `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) { // instead of onerror
  alert("Request failed");
}
```

It might look good, but synchronous calls are used rarely, because they block in-page JavaScript till the loading is complete. In some browsers it becomes impossible to scroll. If a synchronous call takes too much time, the browser may suggest to close the "hanging" webpage.

Many advanced capabilities of `XMLHttpRequest`, like requesting from another domain or specifying a timeout, are unavailable for synchronous requests. Also, as you can see, no progress indication.

Because of all that, synchronous requests are used very sparingly, almost never. We won't talk about them any more.

HTTP-headers

`XMLHttpRequest` allows both to send custom headers and read headers from the response.

There are 3 methods for HTTP-headers:

`setRequestHeader(name, value)`

Sets the request header with the given `name` and `value`.

For instance:

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

> **Headers limitations**
>
> Several headers are managed exclusively by the browser, e.g. `Referer` and `Host`. The full list is in the specification.
>
> `XMLHttpRequest` is not allowed to change them, for the sake of user safety and correctness of the request.

> **Can't remove a header**
>
> Another peculiarity of `XMLHttpRequest` is that one can't undo `setRequestHeader`.
>
> Once the header is set, it's set. Additional calls add information to the header, don't overwrite it.
>
> For instance:
>
> ```
> xhr.setRequestHeader('X-Auth', '123');
> xhr.setRequestHeader('X-Auth', '456');
>
> // the header will be:
> // X-Auth: 123, 456
> ```

`getResponseHeader(name)`

Gets the response header with the given `name` (except `Set-Cookie` and `Set-Cookie2`).

For instance:

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

`getAllResponseHeaders()`

Returns all response headers, except `Set-Cookie` and `Set-Cookie2`.

Headers are returned as a single line, e.g.:

```
Cache-Control: max-age=31536000
Content-Length: 4260
Content-Type: image/png
Date: Sat, 08 Sep 2012 16:53:16 GMT
```

The line break between headers is always `"\r\n"` (doesn't depend on OS), so we can easily split it into individual headers. The separator between the name and the value is always a colon followed by a space `": "`. That's fixed in the specification.

So, if we want to get an object with name/value pairs, we need to throw in a bit JS.

Like this (assuming that if two headers have the same name, then the latter one overwrites the former one):

```
let headers = xhr
  .getAllResponseHeaders()
  .split('\r\n')
  .reduce((result, current) => {
    let [name, value] = current.split(': ');
    result[name] = value;
    return result;
  }, {});

// headers['Content-Type'] = 'image/png'
```

POST, FormData

To make a POST request, we can use the built-in FormData object.

The syntax:

```
let formData = new FormData([form]); // creates an object, optionally fill from <form
formData.append(name, value); // appends a field
```

We create it, optionally fill from a form, `append` more fields if needed, and then:

1. `xhr.open('POST', ...)` – use `POST` method.
2. `xhr.send(formData)` to submit the form to the server.

For instance:

```
<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>
```

The form is sent with `multipart/form-data` encoding.

Or, if we like JSON more, then `JSON.stringify` and send as a string.

Just don't forget to set the header `Content-Type: application/json`, many server-side frameworks automatically decode JSON with it:

```
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);
```

The `.send(body)` method is pretty omnivore. It can send almost any `body`, including `Blob` and `BufferSource` objects.

Upload progress

The `progress` event triggers only on the downloading stage.

That is: if we `POST` something, `XMLHttpRequest` first uploads our data (the request body), then downloads the response.

If we're uploading something big, then we're surely more interested in tracking the upload progress. But `xhr.onprogress` doesn't help here.

There's another object, without methods, exclusively to track upload events: `xhr.upload`.

It generates events, similar to `xhr`, but `xhr.upload` triggers them solely on uploading:

- `loadstart` – upload started.
- `progress` – triggers periodically during the upload.
- `abort` – upload aborted.
- `error` – non-HTTP error.
- `load` – upload finished successfully.
- `timeout` – upload timed out (if `timeout` property is set).
- `loadend` – upload finished with either success or error.

Example of handlers:

```
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}`);
};
```

Here's a real-life example: file upload with progress indication:

```
<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>
```

Cross-origin requests

`XMLHttpRequest` can make cross-origin requests, using the same CORS policy as fetch.

Just like `fetch`, it doesn't send cookies and HTTP-authorization to another origin by default. To enable them, set `xhr.withCredentials` to `true`:

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

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

See the chapter Fetch: Cross-Origin Requests for details about cross-origin headers.

Summary

Typical code of the GET-request with `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)
};
```

There are actually more events, the modern specification lists them (in the lifecycle order):

- `loadstart` – the request has started.
- `progress` – a data packet of the response has arrived, the whole response body at the moment is in `responseText`.
- `abort` – the request was canceled by the call `xhr.abort()`.
- `error` – connection error has occurred, e.g. wrong domain name. Doesn't happen for HTTP-errors like 404.
- `load` – the request has finished successfully.
- `timeout` – the request was canceled due to timeout (only happens if it was set).
- `loadend` – triggers after `load`, `error`, `timeout` or `abort`.

The `error`, `abort`, `timeout`, and `load` events are mutually exclusive. Only one of them may happen.

The most used events are load completion (`load`), load failure (`error`), or we can use a single `loadend` handler and check the properties of the request object `xhr` to see what happened.

We've already seen another event: `readystatechange`. Historically, it appeared long ago, before the specification settled. Nowadays, there's no need to use it, we can replace it with newer events, but it can often be found in older scripts.

If we need to track uploading specifically, then we should listen to same events on `xhr.upload` object.