25th October 2019

Fetch

JavaScript can send network requests to the server and load new information whenever is needed.

For example, we can use a network request to:

- Submit an order,
- Load user information,
- Receive latest updates from the server,
- …etc.

…And all of that without reloading the page!

There's an umbrella term "AJAX" (abbreviated **A**synchronous **J**avaScript **A**nd **X**ML) for network requests from JavaScript. We don't have to use XML though: the term comes from old times, that's why that word is there. You may have heard that term already.

There are multiple ways to send a network request and get information from the server.

The `fetch()` method is modern and versatile, so we'll start with it. It's not supported by old browsers (can be polyfilled), but very well supported among the modern ones.

The basic syntax is:

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

- **`url`** – the URL to access.
- **`options`** – optional parameters: method, headers etc.

Without `options`, that is a simple GET request, downloading the contents of the `url`.

The browser starts the request right away and returns a promise that the calling code should use to get the result.

Getting a response is usually a two-stage process.

First, the `promise`, returned by `fetch`, resolves with an object of the built-in Response class as soon as the server responds with headers.

At this stage we can check HTTP status, to see whether it is successful or not, check headers, but don't have the body yet.

The promise rejects if the `fetch` was unable to make HTTP-request, e.g. network problems, or there's no such site. Abnormal HTTP-statuses, such as 404 or 500 do not cause an error.

We can see HTTP-status in response properties:

- **`status`** – HTTP status code, e.g. 200.
- **`ok`** – boolean, `true` if the HTTP status code is 200-299.

For example:

```
let response = await fetch(url);

if (response.ok) { // if HTTP-status is 200-299
  // get the response body (the method explained below)
  let json = await response.json();
} else {
  alert("HTTP-Error: " + response.status);
}
```

Second, to get the response body, we need to use an additional method call.

`Response` provides multiple promise-based methods to access the body in various formats:

- **`response.text()`** – read the response and return as text,
- **`response.json()`** – parse the response as JSON,
- **`response.formData()`** – return the response as `FormData` object (explained in the next chapter),
- **`response.blob()`** – return the response as Blob (binary data with type),
- **`response.arrayBuffer()`** – return the response as ArrayBuffer (low-level representaion of binary data),
- additionally, `response.body` is a ReadableStream object, it allows to read the body chunk-by-chunk, we'll see an example later.

For instance, let's get a JSON-object with latest commits from GitHub:

```
let url = 'https://api.github.com/repos/javascript-tutorial/en.javascript.inf
let response = await fetch(url);

let commits = await response.json(); // read response body and parse as JSON

alert(commits[0].author.login);
```

Or, the same without `await`, using pure promises syntax:

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

To get the response text, `await response.text()` instead of `.json()`:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/

let text = await response.text(); // read response body as text

alert(text.slice(0, 80) + '...');
```

As a show-case for reading in binary format, let's fetch and show a logo image of “fetch” specification (see chapter Blob for details about operations on `Blob`):

```
let response = await fetch('/article/fetch/logo-fetch.svg');

let blob = await response.blob(); // download as Blob object

// create <img> for it
let img = document.createElement('img');
img.style = 'position:fixed;top:10px;left:10px;width:100px';
document.body.append(img);

// show it
img.src = URL.createObjectURL(blob);

setTimeout(() => { // hide after three seconds
  img.remove();
  URL.revokeObjectURL(img.src);
}, 3000);
```

Important:

We can choose only one body-reading method.

If we've already got the response with `response.text()` , then `response.json()` won't work, as the body content has already been processed.

```
let text = await response.text(); // response body consumed
let parsed = await response.json(); // fails (already consumed)
```

Response headers

The response headers are available in a Map-like headers object in `response.headers` .

It's not exactly a Map, but it has similar methods to get individual headers by name or iterate over them:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/

// get one header
alert(response.headers.get('Content-Type')); // application/json; charset=utf

// iterate over all headers
for (let [key, value] of response.headers) {
  alert(`${key} = ${value}`);
}
```

Request headers

To set a request header in `fetch`, we can use the `headers` option. It has an object with outgoing headers, like this:

```
let response = fetch(protectedUrl, {
  headers: {
    Authentication: 'secret'
  }
});
```

…But there's a list of forbidden HTTP headers that we can't set:

- `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-*`

These headers ensure proper and safe HTTP, so they are controlled exclusively by the browser.

POST requests

To make a `POST` request, or a request with another method, we need to use `fetch` options:

- **`method`** – HTTP-method, e.g. `POST`,
- **`body`** – the request body, one of:
  - a string (e.g. JSON-encoded),
  - `FormData` object, to submit the data as `form/multipart`,
  - `Blob`/`BufferSource` to send binary data,
  - URLSearchParams, to submit the data in `x-www-form-urlencoded` encoding, rarely used.

The JSON format is used most of the time.

For example, this code submits `user` object as JSON:

```
let user = {
  name: 'John',
  surname: 'Smith'
};

let response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});

let result = await response.json();
alert(result.message);
```

Please note, if the request `body` is a string, then `Content-Type` header is set to `text/plain;charset=UTF-8` by default.

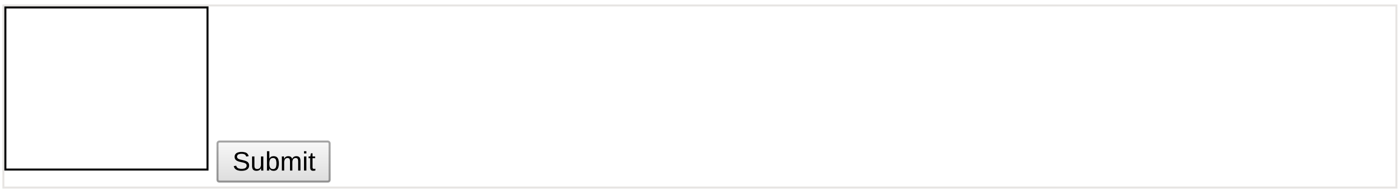
But, as we're going to send JSON, we use `headers` option to send `application/json` instead, the correct `Content-Type` for JSON-encoded data.

Sending an image

We can also submit binary data with `fetch` using `Blob` or `BufferSource` objects.

In this example, there's a `<canvas>` where we can draw by moving a mouse over it. A click on the "submit" button sends the image to server:

```
<body style="margin:0">
  <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></

  <input type="button" value="Submit" onclick="submit()">

  <script>
    canvasElem.onmousemove = function(e) {
      let ctx = canvasElem.getContext('2d');
      ctx.lineTo(e.clientX, e.clientY);
      ctx.stroke();
    };

    async function submit() {
      let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'ima
      let response = await fetch('/article/fetch/post/image', {
        method: 'POST',
        body: blob
      });

      // the server responds with confirmation and the image size
      let result = await response.json();
      alert(result.message);
    }

  </script>
</body>
```



Please note, here we don't set `Content-Type` header manually, because a `Blob` object has a built-in type (here `image/png`, as generated by `toBlob`). For `Blob` objects that type becomes the value of `Content-Type`.

The `submit()` function can be rewritten without `async/await` like this:

```
function submit() {
  canvasElem.toBlob(function(blob) {
    fetch('/article/fetch/post/image', {
      method: 'POST',
      body: blob
    })
      .then(response => response.json())
      .then(result => alert(JSON.stringify(result, null, 2)))
  }, 'image/png');
}
```

Summary

A typical fetch request consists of two `await` calls:

```
let response = await fetch(url, options); // resolves with response headers
let result = await response.json(); // read body as json
```

Or, without `await`:

```
fetch(url, options)
  .then(response => response.json())
  .then(result => /* process result */)
```

Response properties:

- `response.status` – HTTP code of the response,
- `response.ok` – `true` is the status is 200-299.
- `response.headers` – Map-like object with HTTP headers.

Methods to get response body:

- **`response.text()`** – return the response as text,
- **`response.json()`** – parse the response as JSON object,

- **`response.formData()`** – return the response as `FormData` object (form/multipart encoding, see the next chapter),
- **`response.blob()`** – return the response as Blob (binary data with type),
- **`response.arrayBuffer()`** – return the response as ArrayBuffer (low-level binary data),

Fetch options so far:

- `method` – HTTP-method,
- `headers` – an object with request headers (not any header is allowed),
- `body` – the data to send (request body) as `string`, `FormData`, `BufferSource`, `Blob` or `UrlSearchParams` object.

In the next chapters we'll see more options and use cases of `fetch`.

✓ Tasks

Fetch users from GitHub

Create an async function `getUsers(names)`, that gets an array of GitHub logins, fetches the users from GitHub and returns an array of GitHub users.

The GitHub url with user information for the given `USERNAME` is: `https://api.github.com/users/USERNAME`.

There's a test example in the sandbox.

Important details:

1. There should be one `fetch` request per user.
2. Requests shouldn't wait for each other. So that the data arrives as soon as possible.
3. If any request fails, or if there's no such user, the function should return `null` in the resulting array.

Open a sandbox with tests.

solution

Previous lesson Next lesson

Share Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)