Using the Fetch API

The Fetch API provides a JavaScript interface for making HTTP requests and processing the responses.

Fetch is the modern replacement for `XMLHttpRequest`: unlike `XMLHttpRequest`, which uses callbacks, Fetch is promise-based and is integrated with features of the modern web such as service workers and Cross-Origin Resource Sharing (CORS).

With the Fetch API, you make a request by calling `fetch()`, which is available as a global function in both `window` and `worker` contexts. You pass it a `Request` object or a string containing the URL to fetch, along with an optional argument to configure the request.

The `fetch()` function returns a `Promise` which is fulfilled with a `Response` object representing the server's response. You can then check the request status and extract the body of the response in various formats, including text and JSON, by calling the appropriate method on the response.

Here's a minimal function that uses `fetch()` to retrieve some JSON data from a server:

JS

```
async function getData() {
  const url = "https://example.org/products.json";
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error(`Response status: ${response.status}`);
    }

    const json = await response.json();
    console.log(json);
```

```
  } catch (error) {
    console.error(error.message);
  }
}
```

We declare a string containing the URL and then call `fetch()`, passing the URL with no extra options.

The `fetch()` function will reject the promise on some errors, but not if the server responds with an error status like `404`: so we also check the response status and throw if it is not OK.

Otherwise, we fetch the response body content as JSON by calling the `json()` method of `Response`, and log one of its values. Note that like `fetch()` itself, `json()` is asynchronous, as are all the other methods to access the response body content.

In the rest of this page we'll look in more detail at the different stages of this process.

Making a request

To make a request, call `fetch()`, passing in:

1. a definition of the resource to fetch. This can be any one of:
    - a string containing the URL
    - an object, such an instance of `URL`, which has a stringifier that produces a string containing the URL
    - a `Request` instance
2. optionally, an object containing options to configure the request.

In this section we'll look at some of the most commonly-used options. To read about all the options that can be given, see the `fetch()` reference page.

Setting the method

By default, `fetch()` makes a `GET` request, but you can use the `method` option to use a different request method:

```
JS
```

```
const response = await fetch("https://example.org/post", {
  method: "POST",
  // ...
});
```

If the `mode` option is set to `no-cors`, then `method` must be one of `GET`, `POST` or `HEAD`.

Setting a body

The request body is the payload of the request: it's the thing the client is sending to the server. You cannot include a body with `GET` requests, but it's useful for requests that send content to the server, such as `POST` or `PUT` requests. For example, if you want to upload a file to the server, you might make a `POST` request and include the file as the request body.

To set a request body, pass it as the `body` option:

```
JS
```

```
const response = await fetch("https://example.org/post", {
  body: JSON.stringify({ username: "example" }),
  // ...
});
```

You can supply the body as an instance of any of the following types:

- a string
- `ArrayBuffer`
- `TypedArray`
- `DataView`

- `Blob`
- `File`
- `URLSearchParams`
- `FormData`

Note that just like response bodies, request bodies are streams, and making the request reads the stream, so if a request contains a body, you can't make it twice:

JS

```
const request = new Request("https://example.org/post", {
  method: "POST",
  body: JSON.stringify({ username: "example" }),
});

const response1 = await fetch(request);
console.log(response1.status);

// Will throw: "Body has already been consumed."
const response2 = await fetch(request);
console.log(response2.status);
```

Instead, you would need to create a clone of the request before sending it:

JS

```
const request1 = new Request("https://example.org/post", {
  method: "POST",
  body: JSON.stringify({ username: "example" }),
});

const request2 = request1.clone();

const response1 = await fetch(request1);
console.log(response1.status);

const response2 = await fetch(request2);
console.log(response2.status);
```

See Locked and disturbed streams for more information.

Setting headers

Request headers give the server information about the request: for example, the `Content-Type` header tells the server the format of the request's body. Many headers are set automatically by the browser and can't be set by a script: these are called Forbidden header names.

To set request headers, assign them to the `headers` option.

You can pass an object literal here containing `header-name: header-value` properties:

JS

```
const response = await fetch("https://example.org/post", {
  headers: {
    "Content-Type": "application/json",
  },
  // .,.
});
```

Alternatively, you can construct a `Headers` object, add headers to that object using `Headers.append()`, then assign the `Headers` object to the `headers` option:

JS

```
const myHeaders = new Headers();
myHeaders.append("Content-Type", "application/json");

const response = await fetch("https://example.org/post", {
  headers: myHeaders,
  // .,.
});
```

If the `mode` option is set to `no-cors`, you can only set CORS-safelisted request headers.

Making POST requests

We can combine the `method`, `body`, and `headers` options to make a POST request:

```JS
const myHeaders = new Headers();
myHeaders.append("Content-Type", "application/json");

const response = await fetch("https://example.org/post", {
  method: "POST",
  body: JSON.stringify({ username: "example" }),
  headers: myHeaders,
});
```

Making cross-origin requests

Whether a request can be made cross-origin or not is determined by the value of the `mode` option. This may take one of three values: `cors`, `no-cors`, or `same-origin`.

- By default, `mode` is set to `cors`, meaning that if the request is cross-origin then it will use the Cross-Origin Resource Sharing (CORS) mechanism. This means that:
  - if the request is a simple request, then the request will always be sent, but the server must respond with the correct `Access-Control-Allow-Origin` header or the browser will not share the response with the caller.
  - if the request is not a simple request, then the browser will send a preflighted request to check that the server understands CORS and allows the request, and the real request will not be sent unless the server responds to the preflighted request with the appropriate CORS headers.
- Setting `mode` to `same-origin` disallows cross-origin requests completely.
- Setting `mode` to `no-cors` means the request must be a simple request, which restricts the headers that may be set, and restricts methods to `GET`, `HEAD`, and `POST`.

Including credentials

Credentials are cookies, TLS client certificates, or authentication headers containing a

username and password.

To control whether or not the browser sends credentials, set the `credentials` option, which can take one of the following three values:

- `omit`: never send credentials in the request or include credentials in the response.
- `same-origin` (the default): only send and include credentials for same-origin requests.
- `include`: always include credentials, even cross-origin.

Note that if a cookie's `SameSite` attribute is set to `Strict` or `Lax`, then the cookie will not be sent cross-site, even if `credentials` is set to `include`.

Including credentials in cross-origin requests can make a site vulnerable to CSRF attacks, so even if `credentials` is set to `include`, the server must also agree to their inclusion by including the `Access-Control-Allow-Credentials` header in its response. Additionally, in this situation the server must explicitly specify the client's origin in the `Access-Control-Allow-Origin` response header (that is, `*` is not allowed).

This means that if `credentials` is set to `include` and the request is cross-origin, then:

- If the request is a simple request, then the request will be sent with credentials, but the server must set the `Access-Control-Allow-Credentials` and `Access-Control-Allow-Origin` response headers, or the browser will return a network error to the caller. If the server does set the correct headers, then the response, including credentials, will be delivered to the caller.
- If the request is not a simple request, then the browser will send a preflighted request without credentials, and the server must set the `Access-Control-Allow-Credentials` and `Access-Control-Allow-Origin` response headers, or the browser will return a network error to the caller. If the server does set the correct headers, then the browser will follow up with the real request, including credentials, and will deliver the real response, including credentials, to the caller.

Creating a `Request` object

The `Request()` constructor takes the same arguments as `fetch()` itself. This means that instead of passing options into `fetch()`, you can pass the same options to the `Request()` constructor, and then pass that object to `fetch()`.

For example, we can make a POST request by passing options into `fetch()` using code like this:

JS

```
const myHeaders = new Headers();
myHeaders.append("Content-Type", "application/json");

const response = await fetch("https://example.org/post", {
  method: "POST",
  body: JSON.stringify({ username: "example" }),
  headers: myHeaders,
});
```

However, we could rewrite this to pass the same arguments to the `Request()` constructor:

JS

```
const myHeaders = new Headers();
myHeaders.append("Content-Type", "application/json");

const myRequest = new Request("https://example.org/post", {
  method: "POST",
  body: JSON.stringify({ username: "example" }),
  headers: myHeaders,
});

const response = await fetch(myRequest);
```

This also means that you can create a request from another request, while changing some of its properties using the second argument:

JS

```
async function post(request) {
  try {
    const response = await fetch(request);
    const result = await response.json();
    console.log("Success:", result);
  } catch (error) {
    console.error("Error:", error);
  }
}

const request1 = new Request("https://example.org/post", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify({ username: "example1" }),
});

const request2 = new Request(request1, {
  body: JSON.stringify({ username: "example2" }),
});

post(request1);
post(request2);
```

Canceling a request

To make a request cancelable, create an `AbortController`, and assign its `AbortSignal` to the request's `signal` property.

To cancel the request, call the controller's `abort()` method. The `fetch()` call will reject the promise with an `AbortError` exception.

JS

```
const controller = new AbortController();

const fetchButton = document.querySelector("#fetch");
fetchButton.addEventListener("click", async () => {
```

```
  try {
    console.log("Starting fetch");
    const response = await fetch("https://example.org/get", {
      signal: controller.signal,
    });
    console.log(`Response: ${response.status}`);
  } catch (e) {
    console.error(`Error: ${e}`);
  }
});

const cancelButton = document.querySelector("#cancel");
cancelButton.addEventListener("click", () => {
  controller.abort();
  console.log("Canceled fetch");
});
```

If the request is aborted after the `fetch()` call has been fulfilled but before the response body has been read, then attempting to read the response body will reject with an `AbortError` exception.

JS

```
async function get() {
  const controller = new AbortController();
  const request = new Request("https://example.org/get", {
    signal: controller.signal,
  });

  const response = await fetch(request);
  controller.abort();
  // The next line will throw `AbortError`
  const text = await response.text();
  console.log(text);
}
```

Handling the response

As soon as the browser has received the response status and headers from the server (and potentially before the response body itself has been received), the promise

returned by `fetch()` is fulfilled with a `Response` object.

Checking response status

The promise returned by `fetch()` will reject on some errors, such as a network error or a bad scheme. However, if the server responds with an error like `404`, then `fetch()` fulfills with a `Response`, so we have to check the status before we can read the response body.

The `Response.status` property tells us the numerical status code, and the `Response.ok` property returns `true` if the status is in the 200 range.

A common pattern is to check the value of `ok` and throw if it is `false`:

JS

```
async function getData() {
  const url = "https://example.org/products.json";
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error(`Response status: ${response.status}`);
    }
    // ...
  } catch (error) {
    console.error(error.message);
  }
}
```

Checking the response type

Responses have a `type` property that can be one of the following:

- `basic`: the request was a same-origin request.
- `cors`: the request was a cross-origin CORS request.
- `opaque`: the request was a cross-origin simple request made with the `no-cors` mode.
- `opaqueredirect`: the request set the `redirect` option to `manual`, and the server returned a redirect status.

The type determines the possible contents of the response, as follows:

- Basic responses exclude response headers from the Forbidden response header name list.
- CORS responses include only response headers from the CORS-safelisted response header list.
- Opaque responses and opaque redirect responses have a `status` of `0`, an empty header list, and a `null` body.

Checking headers

Just like the request, the response has a `headers` property which is a `Headers` object, and this contains any response headers that are exposed to scripts, subject to the exclusions made based on the response type.

A common use case for this is to check the content type before trying to read the body:

JS

```
async function fetchJSON(request) {
  try {
    const response = await fetch(request);
    const contentType = response.headers.get("content-type");
    if (!contentType || !contentType.includes("application/json")) {
      throw new TypeError("Oops, we haven't got JSON!");
    }
    // Otherwise, we can read the body as JSON
  } catch (error) {
    console.error("Error:", error);
  }
}
```

Reading the response body

The `Response` interface provides a number of methods to retrieve the entire body contents in a variety of different formats:

- `Response.arrayBuffer()`
- `Response.blob()`
- `Response.formData()`
- `Response.json()`
- `Response.text()`

These are all asynchronous methods, returning a `Promise` which will be fulfilled with the body content.

In this example, we fetch an image and read it as a `Blob`, which we can then use to create an object URL:

JS

```
const image = document.querySelector("img");

const url = "flowers.jpg";

async function setImage() {
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error(`Response status: ${response.status}`);
    }
    const blob = await response.blob();
    const objectURL = URL.createObjectURL(blob);
    image.src = objectURL;
  } catch (e) {
    console.error(e);
  }
}
```

The method will throw an exception if the response body is not in the appropriate format: for example, if you call `json()` on a response that can't be parsed as JSON.

Streaming the response body

Request and response bodies are actually `ReadableStream` objects, and whenever you read them, you're streaming the content. This is good for memory efficiency, because the browser doesn't have to buffer the entire response in memory before the caller retrieves it using a method like `json()`.

This also means that the caller can process the content incrementally as it is received.

For example, consider a `GET` request that fetches a large text file and processes it in some way, or displays it to the user:

JS

```
const url = "https://www.example.org/a-large-file.txt";

async function fetchText(url) {
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error(`Response status: ${response.status}`);
    }

    const text = await response.text();
    console.log(text);
  } catch (e) {
    console.error(e);
  }
}
```

If we use `Response.text()`, as above, we must wait until the whole file has been received before we can process any of it.

If we stream the response instead, we can process chunks of the body as they are received from the network:

JS

```
const url = "https://www.example.org/a-large-file.txt";
```

```
async function fetchTextAsStream(url) {
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error(`Response status: ${response.status}`);
    }

    const stream = response.body.pipeThrough(new TextDecoderStream());
    for await (const value of stream) {
      console.log(value);
    }
  } catch (e) {
    console.error(e);
  }
}
```

In this example, we iterate asynchronously over the stream, processing each chunk as it arrives.

Note that when you access the body directly like this, you get the raw bytes of the response and must transform it yourself. In this case we call `ReadableStream.pipeThrough()` to pipe the response through a `TextDecoderStream`, which decodes the UTF-8-encoded body data as text.

Processing a text file line by line

In the example below, we fetch a text resource and process it line by line, using a regular expression to look for line endings. For simplicity, we assume the text is UTF-8, and don't handle fetch errors:

JS

```
async function* makeTextFileLineIterator(fileURL) {
  const response = await fetch(fileURL);
  const reader = response.body.pipeThrough(new TextDecoderStream()).getReader();

  let { value: chunk, done: readerDone } = await reader.read();
  chunk = chunk || "";
```

```
  const newline = /\r?\n/gm;
  let startIndex = 0;
  let result;

  while (true) {
    const result = newline.exec(chunk);
    if (!result) {
      if (readerDone) break;
      const remainder = chunk.substr(startIndex);
      ({ value: chunk, done: readerDone } = await reader.read());
      chunk = remainder + (chunk || "");
      startIndex = newline.lastIndex = 0;
      continue;
    }
    yield chunk.substring(startIndex, result.index);
    startIndex = newline.lastIndex;
  }

  if (startIndex < chunk.length) {
    // Last line didn't end in a newline char
    yield chunk.substring(startIndex);
  }
}

async function run(urlOfFile) {
  for await (const line of makeTextFileLineIterator(urlOfFile)) {
    processLine(line);
  }
}

function processLine(line) {
  console.log(line);
}

run("https://www.example.org/a-large-file.txt");
```

Locked and disturbed streams

The consequences of request and response bodies being streams are that:

- if a reader has been attached to a stream using `ReadableStream.getReader()`, then the

stream is *locked*, and nothing else can read the stream.

- if any content has been read from the stream, then the stream is *disturbed*, and nothing else can read from the stream.

This means it's not possible to read the same response (or request) body more than once:

JS

```
async function getData() {
  const url = "https://example.org/products.json";
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error(`Response status: ${response.status}`);
    }

    const json1 = await response.json();
    const json2 = await response.json(); // will throw
  } catch (error) {
    console.error(error.message);
  }
}
```

If you do need to read the body more than once, you must call `Response.clone()` before reading the body:

JS

```
async function getData() {
  const url = "https://example.org/products.json";
  try {
    const response1 = await fetch(url);
    if (!response1.ok) {
      throw new Error(`Response status: ${response1.status}`);
    }

    const response2 = response1.clone();

    const json1 = await response1.json();
```

```
    const json2 = await response2.json();
  } catch (error) {
    console.error(error.message);
  }
}
```

This is a common pattern when implementing an offline cache with service workers. The service worker wants to return the response to the app, but also to cache the response. So it clones the response, returns the original, and caches the clone:

JS

```
async function cacheFirst(request) {
  const cachedResponse = await caches.match(request);
  if (cachedResponse) {
    return cachedResponse;
  }
  try {
    const networkResponse = await fetch(request);
    if (networkResponse.ok) {
      const cache = await caches.open("MyCache_1");
      cache.put(request, networkResponse.clone());
    }
    return networkResponse;
  } catch (error) {
    return Response.error();
  }
}

self.addEventListener("fetch", (event) => {
  if (precachedResources.includes(url.pathname)) {
    event.respondWith(cacheFirst(event.request));
  }
});
```

See also

- Service Worker API
- Streams API

- CORS
- HTTP
- Fetch examples on GitHub

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on Jul 4, 2024 by MDN contributors.

View this page on GitHub • Report a problem with this content