# Using the Fetch API

The Fetch API provides a JavaScript interface for accessing and manipulating parts of the protocol, such as requests and responses. It also provides a global `fetch()` method that provides an easy, logical way to fetch resources asynchronously across the network.

Unlike `XMLHttpRequest` that is a callback-based API, Fetch is promise-based and provides a better alternative that can be easily used in service workers. Fetch also integrates advanced HTTP concepts such as CORS and other extensions to HTTP.

A basic fetch request looks like this:

```
async function logJSONData() {
  const response = await fetch("http://example.com/movies.json");
  const jsonData = await response.json();
  console.log(jsonData);
}
```

Here we are fetching a JSON file across the network and printing it to the console. The simplest use of `fetch()` takes one argument — the path to the resource you want to fetch — and does not directly return the JSON response body but instead returns a promise that resolves with a `Response` object.

The `Response` object, in turn, does not directly contain the actual JSON response body but is instead a representation of the entire HTTP response. So, to extract the JSON body content from the `Response` object, we use the `json()` method, which returns a second promise that resolves with the result of parsing the response body text as JSON.

> **Note:** See the Body section for similar methods to extract other types of body content.

Fetch requests are controlled by the `connect-src` directive of Content Security Policy rather than the directive of the resources it's retrieving.

## Supplying request options

The `fetch()` method can optionally accept a second parameter, an `init` object that allows you to control a number of different settings:

See `fetch()` for the full options available, and more details.

```
// Example POST method implementation:
async function postData(url = "", data = {}) {
  // Default options are marked with *
  const response = await fetch(url, {
    method: "POST", // *GET, POST, PUT, DELETE, etc.
    mode: "cors", // no-cors, *cors, same-origin
    cache: "no-cache", // *default, no-cache, reload, force-cache, only-if-cached
    credentials: "same-origin", // include, *same-origin, omit
    headers: {
      "Content-Type": "application/json",
      // 'Content-Type': 'application/x-www-form-urlencoded',
    },
    redirect: "follow", // manual, *follow, error
    referrerPolicy: "no-referrer", // no-referrer, *no-referrer-when-downgrade, origin,
origin-when-cross-origin, same-origin, strict-origin, strict-origin-when-cross-origin,
unsafe-url
    body: JSON.stringify(data), // body data type must match "Content-Type" header
  });
  return response.json(); // parses JSON response into native JavaScript objects
}

postData("https://example.com/answer", { answer: 42 }).then((data) => {
  console.log(data); // JSON data parsed by `data.json()` call
});
```

Note that `mode: "no-cors"` only allows a limited set of headers in the request:

- `Accept`
- `Accept-Language`
- `Content-Language`
- `Content-Type` with a value of `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`

## Aborting a fetch

To abort incomplete `fetch()` operations, use the `AbortController` and `AbortSignal` interfaces.

```
const controller = new AbortController();
const signal = controller.signal;
const url = "video.mp4";

const downloadBtn = document.querySelector("#download");
const abortBtn = document.querySelector("#abort");

downloadBtn.addEventListener("click", async () => {
  try {
    const response = await fetch(url, { signal });
    console.log("Download complete", response);
  } catch (error) {
    console.error(`Download error: ${error.message}`);
  }
});

abortBtn.addEventListener("click", () => {
  controller.abort();
  console.log("Download aborted");
});
```

## Sending a request with credentials included

To cause browsers to send a request with credentials included on both same-origin and cross-origin calls, add `credentials: 'include'` to the `init` object you pass to the `fetch()` method.

```
fetch("https://example.com", {
  credentials: "include",
});
```

> **Note:** `Access-Control-Allow-Origin` is prohibited from using a wildcard for requests with `credentials: 'include'`. In such cases, the exact origin must be provided; even if you are using a CORS unblocker extension, the requests will still fail.

> **Note:** Browsers should not send credentials in *preflight requests* irrespective of this setting. For more information see: CORS Requests with credentials.

If you only want to send credentials if the request URL is on the same origin as the calling script, add `credentials: 'same-origin'`.

```
// The calling script is on the origin 'https://example.com'

fetch("https://example.com", {
  credentials: "same-origin",
});
```

To instead ensure browsers don't include credentials in the request, use `credentials: 'omit'`.

```
fetch("https://example.com", {
  credentials: "omit",
});
```

## Uploading JSON data

Use `fetch()` to POST JSON-encoded data.

```
async function postJSON(data) {
  try {
    const response = await fetch("https://example.com/profile", {
      method: "POST", // or 'PUT'
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(data),
    });

    const result = await response.json();
    console.log("Success:", result);
  } catch (error) {
    console.error("Error:", error);
  }
}

const data = { username: "example" };
postJSON(data);
```

## Uploading a file

Files can be uploaded using an HTML `<input type="file" />` input element, `FormData()` and `fetch()`.

```
async function upload(formData) {
  try {
    const response = await fetch("https://example.com/profile/avatar", {
      method: "PUT",
      body: formData,
    });
    const result = await response.json();
    console.log("Success:", result);
  } catch (error) {
    console.error("Error:", error);
  }
}
```

```
const formData = new FormData();
const fileField = document.querySelector('input[type="file"]');

formData.append("username", "abc123");
formData.append("avatar", fileField.files[0]);

upload(formData);
```

## Uploading multiple files

Files can be uploaded using an HTML `<input type="file" multiple />` input element, `FormData()` and `fetch()`.

```
async function uploadMultiple(formData) {
  try {
    const response = await fetch("https://example.com/posts", {
      method: "POST",
      body: formData,
    });
    const result = await response.json();
    console.log("Success:", result);
  } catch (error) {
    console.error("Error:", error);
  }
}

const photos = document.querySelector('input[type="file"][multiple]');
const formData = new FormData();

formData.append("title", "My Vegas Vacation");

for (const [i, photo] of Array.from(photos.files).entries()) {
  formData.append(`photos_${i}`, photo);
}

uploadMultiple(formData);
```

## Processing a text file line by line

The chunks that are read from a response are not broken neatly at line boundaries and are Uint8Arrays, not strings. If you want to fetch a text file and process it line by line, it is up to you to handle these complications. The following example shows one way to do this by creating a line iterator (for simplicity, it assumes the text is UTF-8, and doesn't handle fetch errors).

```
async function* makeTextFileLineIterator(fileURL) {
  const utf8Decoder = new TextDecoder("utf-8");
  const response = await fetch(fileURL);
  const reader = response.body.getReader();
  let { value: chunk, done: readerDone } = await reader.read();
  chunk = chunk ? utf8Decoder.decode(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 ? utf8Decoder.decode(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.substr(startIndex);
  }
}

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

run();
```

## Checking that the fetch was successful

A `fetch()` promise will reject with a `TypeError` when a network error is encountered or CORS is misconfigured on the server-side, although this usually means permission issues or similar — a 404 does not constitute a network error, for example. An accurate check for a successful `fetch()` would include checking that the promise resolved, then checking that the `Response.ok` property has a value of true. The code would look something like this:

```
async function fetchImage() {
  try {
    const response = await fetch("flowers.jpg");
    if (!response.ok) {
      throw new Error("Network response was not OK");
    }
    const myBlob = await response.blob();
    myImage.src = URL.createObjectURL(myBlob);
  } catch (error) {
    console.error("There has been a problem with your fetch operation:", error);
  }
}
```

## Supplying your own request object

Instead of passing a path to the resource you want to request into the `fetch()` call, you can create a request object using the `Request()` constructor, and pass that in as a `fetch()` method argument:

```
async function fetchImage(request) {
  try {
    const response = await fetch(request);
    if (!response.ok) {
      throw new Error("Network response was not OK");
    }
    const myBlob = await response.blob();
    myImage.src = URL.createObjectURL(myBlob);
  } catch (error) {
    console.error("Error:", error);
  }
}

const myHeaders = new Headers();

const myRequest = new Request("flowers.jpg", {
  method: "GET",
  headers: myHeaders,
  mode: "cors",
  cache: "default",
});

fetchImage(myRequest);
```

`Request()` accepts exactly the same parameters as the `fetch()` method. You can even pass in an existing request object to create a copy of it:

```
const anotherRequest = new Request(myRequest, myInit);
```

This is pretty useful, as request and response bodies can only be used once. Making a copy like this allows you to effectively use the request/response again while varying the `init` options if desired. The copy must be made before the body is read.

> **Note:** There is also a `clone()` method that creates a copy. Both methods of creating a copy will fail if the body of the original request or response has already been read, but reading the body of a cloned response or request will not cause it to be marked as read in the original.

## Headers

mdn web docs

The `Headers` interface allows you to create your own headers object via the `Headers()` constructor. A headers object is a simple multi-map of names to values:

```
const content = "Hello World";
const myHeaders = new Headers();
myHeaders.append("Content-Type", "text/plain");
myHeaders.append("Content-Length", content.length.toString());
myHeaders.append("X-Custom-Header", "ProcessThisImmediately");
```

The same can be achieved by passing an array of arrays or an object literal to the constructor:

```
const myHeaders = new Headers({
  "Content-Type": "text/plain",
  "Content-Length": content.length.toString(),
  "X-Custom-Header": "ProcessThisImmediately",
});
```

The contents can be queried and retrieved:

```
console.log(myHeaders.has("Content-Type")); // true
console.log(myHeaders.has("Set-Cookie")); // false
myHeaders.set("Content-Type", "text/html");
myHeaders.append("X-Custom-Header", "AnotherValue");

console.log(myHeaders.get("Content-Length")); // 11
console.log(myHeaders.get("X-Custom-Header")); // ['ProcessThisImmediately',
'AnotherValue']

myHeaders.delete("X-Custom-Header");
console.log(myHeaders.get("X-Custom-Header")); // null
```

Some of these operations are only useful in `ServiceWorkers`, but they provide a much nicer API for manipulating headers.

All of the Headers methods throw a `TypeError` if a header name is used that is not a valid HTTP Header name. The mutation operations will throw a `TypeError` if there is an immutable guard (see below). Otherwise, they fail silently. For example:

```
const myResponse = Response.error();
try {
  myResponse.headers.set("Origin", "http://mybank.com");
} catch (e) {
  console.log("Cannot pretend to be a bank!");
}
```

A good use case for headers is checking whether the content type is correct before you process it further. For example:

```
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!");
    }
    const jsonData = await response.json();
    // process your data further
  } catch (error) {
    console.error("Error:", error);
  }
}
```

## Guard

Since headers can be sent in requests and received in responses, and have various limitations about what information can and should be mutable, headers' objects have a *guard* property. This is not exposed to the Web, but it affects which mutation operations are allowed on the headers object.

Possible guard values are:

- `none` : default.
- `request` : guard for a headers object obtained from a request ( `Request.headers` ).
- `request-no-cors` : guard for a headers object obtained from a request created with `Request.mode` `no-cors` .
- `response` : guard for a headers object obtained from a response ( `Response.headers` ).
- `immutable` : guard that renders a headers object read-only; mostly used for ServiceWorkers.

> **Note:** You may not append or set the `Content-Length` header on a guarded headers object for a `response` . Similarly, inserting `Set-Cookie` into a response header is not allowed: ServiceWorkers are not allowed to set cookies via synthesized responses.

## Response objects

As you have seen above, `Response` instances are returned when `fetch()` promises are resolved.

The most common response properties you'll use are:

- `Response.status` — An integer (default value 200) containing the response status code.
- `Response.statusText` — A string (default value ""), which corresponds to the HTTP status code message. Note that HTTP/2 does not support status messages.
- `Response.ok` — seen in use above, this is a shorthand for checking that status is in the range 200-299 inclusive. This returns a boolean value.

They can also be created programmatically via JavaScript, but this is only really useful in `ServiceWorkers` , when you are providing a custom response to a received request using a `respondWith()` method:

```
const myBody = new Blob();
```

```
addEventListener("fetch", (event) => {
  // ServiceWorker intercepting a fetch
  event.respondWith(
    new Response(myBody, {
      headers: { "Content-Type": "text/plain" },
    })
  );
});
```

The `Response()` constructor takes two optional arguments — a body for the response, and an init object (similar to the one that `Request()` accepts.)

> **Note:** The static method `error()` returns an error response. Similarly, `redirect()` returns a response resulting in a redirect to a specified URL. These are also only relevant to Service Workers.

## Body

Both requests and responses may contain body data. A body is an instance of any of the following types:

- `ArrayBuffer`
- `TypedArray` (Uint8Array and friends)
- `DataView`
- `Blob`
- `File`
- `String`, or a string literal
- `URLSearchParams`
- `FormData`

The `Request` and `Response` interfaces share the following methods to extract a body. These all return a promise that is eventually resolved with the actual content.

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

This makes usage of non-textual data much easier than it was with XHR.

Request bodies can be set by passing body parameters:

```
const form = new FormData(document.getElementById("login-form"));
fetch("/login", {
  method: "POST",
  body: form,
});
```

Both request and response (and by extension the `fetch()` function), will try to intelligently determine the content type. A request will also automatically set a `Content-Type` header if none is set in the dictionary.

## Feature detection

Fetch API support can be detected by checking for the existence of `Headers`, `Request`, `Response` or `fetch()` on the `Window` or `Worker` scope. For example:

```
if (window.fetch) {
  // run my fetch request here
} else {
  // do something with XMLHttpRequest?
}
```

## Differences from `jQuery.ajax()`

The `fetch` specification differs from `jQuery.ajax()` in the following significant ways:

- The promise returned from `fetch()` won't reject on HTTP errors even if the response is an HTTP 404 or 500. Instead, as soon as the server responds with headers, the promise will resolve (with the `ok` property of the response set to `false` if the response isn't in the range 200–299). The promise will only reject on network failure or if anything prevented the request from completing.
- Unless `fetch()` is called with the `credentials` option set to `include`, `fetch()`:
  - won't send cookies in cross-origin requests
  - won't set any cookies sent back in cross-origin responses
  - As of August 2018, the default credentials policy changed to same-origin.

## See also

- ServiceWorker API
- HTTP access control (CORS)
- HTTP
- Fetch polyfill
- Fetch examples on GitHub

This page was last modified on Apr 3, 2023 by MDN contributors.