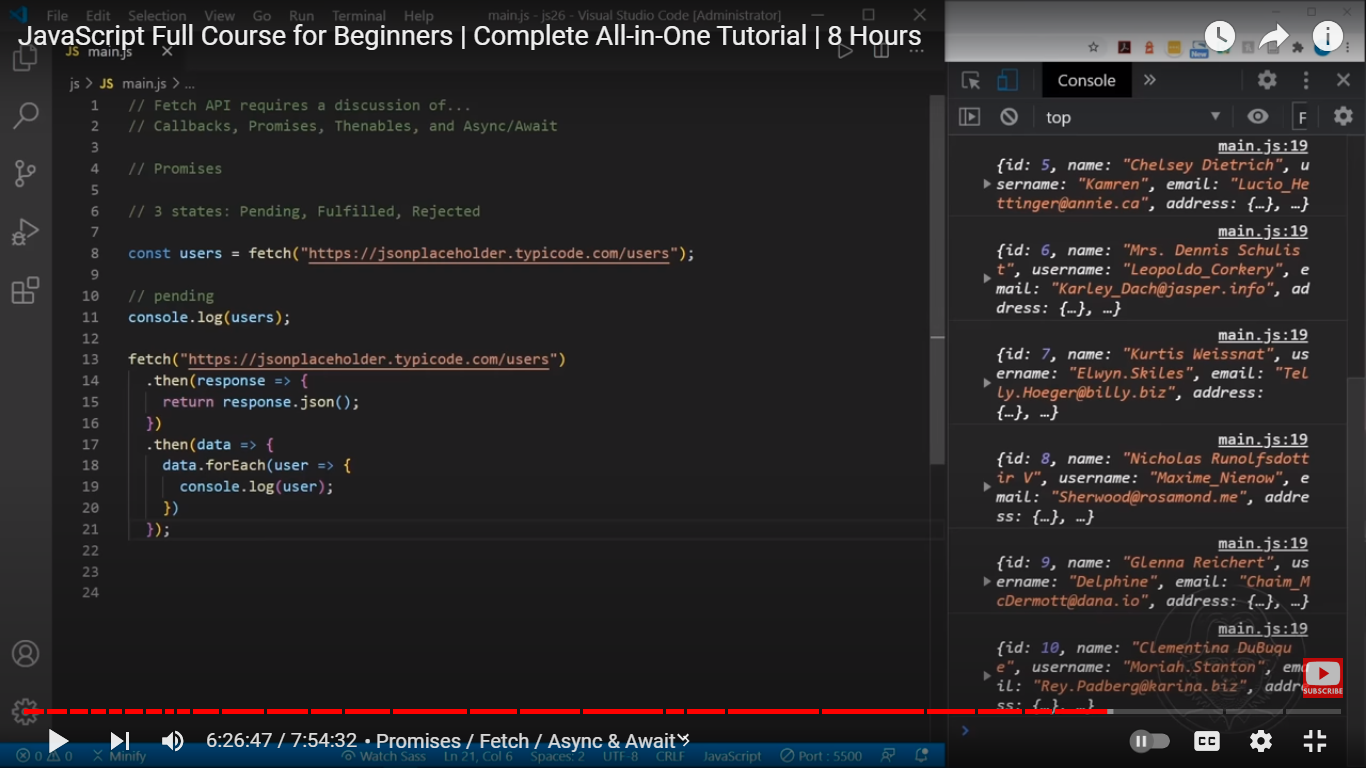
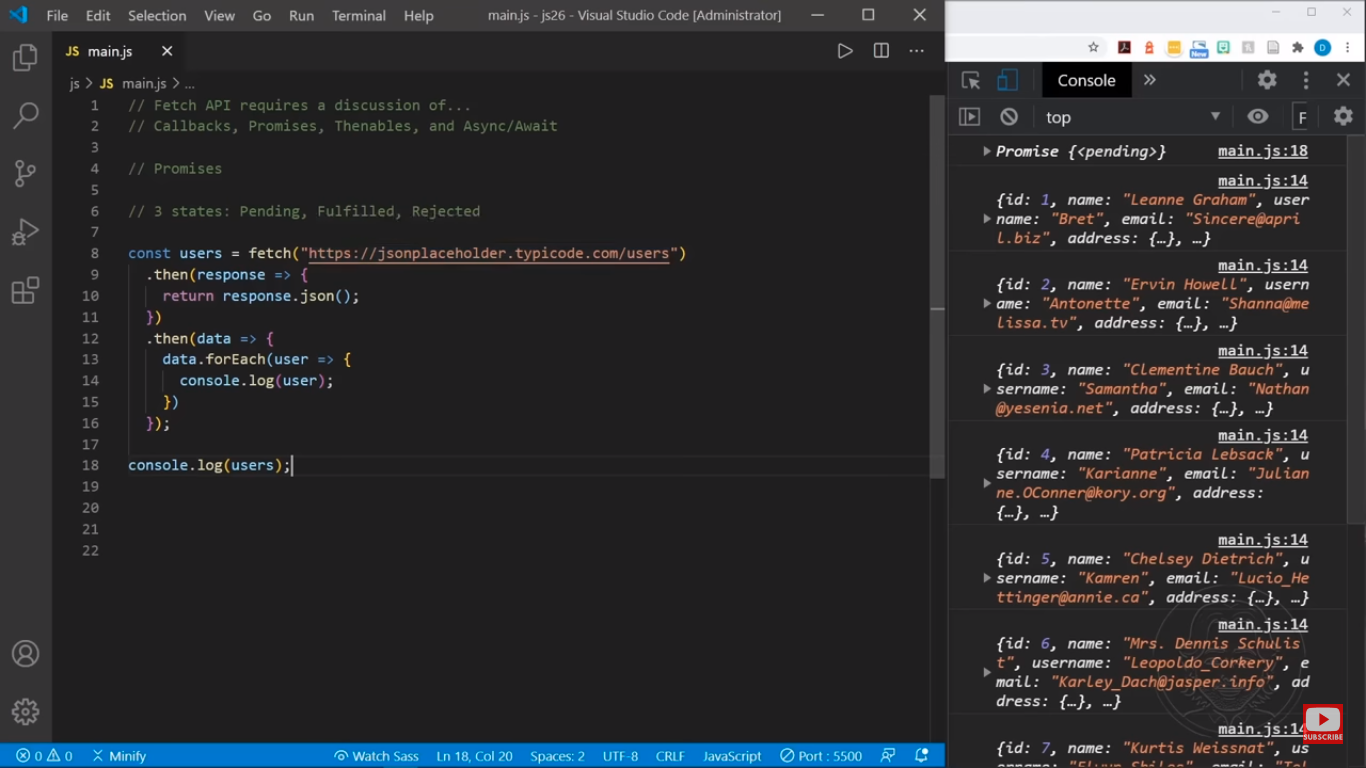
Fetch API

For in depth detail: <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch>

fetch() .then() .json() function will return promises object

pending: the promise has been created, and the asynchronous function it's associated with has not succeeded or failed yet. This is the state your promise is in when it's returned from a call to fetch(), and the request is still being made.





Fetch always return promises object, in this example the .then() method will be execute but console.log() method is executed because the then() method is wait to fetch an api from another program.

The [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) provides a JavaScript interface for accessing and manipulating parts of the [protocol](https://developer.mozilla.org/en-US/docs/Glossary/Protocol), such as requests and responses. It also provides a global [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch) method that provides an easy, logical way to fetch resources asynchronously across the network.

Unlike [XMLHttpRequest](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) that is a callback-based API, Fetch is promise-based and provides a better alternative that can be easily used in [service workers](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API). Fetch also integrates advanced HTTP concepts such as [CORS](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS) and other extensions to HTTP.

A basic fetch request looks like this:

JSCopy to Clipboard

async function logMovies() {

const response = await fetch("http://example.com/movies.json");

const movies = await response.json();

console.log(movies);

}

Here we are fetching a JSON file across the network, parsing it, and printing the data 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](https://developer.mozilla.org/en-US/docs/Web/API/Response) object.

The [Response](https://developer.mozilla.org/en-US/docs/Web/API/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](https://developer.mozilla.org/en-US/docs/Web/API/Response) object, we use the [json()](https://developer.mozilla.org/en-US/docs/Web/API/Response/json) method, which returns a second promise that resolves with the result of parsing the response body text as JSON.

[**Supplying request options**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#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()](https://developer.mozilla.org/en-US/docs/Web/API/fetch) for the full options available, and more details.

JSCopy to Clipboard

// 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**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#aborting_a_fetch)

To abort incomplete fetch() operations, use the [AbortController](https://developer.mozilla.org/en-US/docs/Web/API/AbortController) and [AbortSignal](https://developer.mozilla.org/en-US/docs/Web/API/AbortSignal) interfaces.

JSCopy to Clipboard

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**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#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.

JSCopy to Clipboard

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](https://developer.mozilla.org/en-US/docs/Web/HTTP/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'.

JSCopy to Clipboard

// 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'.

JSCopy to Clipboard

fetch("https://example.com", {

credentials: "omit",

});

[**Uploading JSON data**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#uploading_json_data)

Use [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch) to POST JSON-encoded data.

JSCopy to Clipboard

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**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#uploading_a_file)

Files can be uploaded using an HTML <input type="file" /> input element, [FormData()](https://developer.mozilla.org/en-US/docs/Web/API/FormData/FormData) and [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch).

JSCopy to Clipboard

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**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#uploading_multiple_files)

Files can be uploaded using an HTML <input type="file" multiple /> input element, [FormData()](https://developer.mozilla.org/en-US/docs/Web/API/FormData/FormData) and [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch).

JSCopy to Clipboard

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**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#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).

JSCopy to Clipboard

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**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#checking_that_the_fetch_was_successful)

A [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch) promise will reject with a [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/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](https://developer.mozilla.org/en-US/docs/Web/API/Response/ok) property has a value of true. The code would look something like this:

JSCopy to Clipboard

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**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#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()](https://developer.mozilla.org/en-US/docs/Web/API/Request/Request) constructor, and pass that in as a fetch() method argument:

JSCopy to Clipboard

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:

JSCopy to Clipboard

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()](https://developer.mozilla.org/en-US/docs/Web/API/Request/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**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#headers)

The [Headers](https://developer.mozilla.org/en-US/docs/Web/API/Headers) interface allows you to create your own headers object via the [Headers()](https://developer.mozilla.org/en-US/docs/Web/API/Headers/Headers) constructor. A headers object is a simple multi-map of names to values:

JSCopy to Clipboard

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:

JSCopy to Clipboard

const myHeaders = new Headers({

"Content-Type": "text/plain",

"Content-Length": content.length.toString(),

"X-Custom-Header": "ProcessThisImmediately",

});

The contents can be queried and retrieved:

JSCopy to Clipboard

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](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API), 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](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#guard)). Otherwise, they fail silently. For example:

JSCopy to Clipboard

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:

JSCopy to Clipboard

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**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#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](https://developer.mozilla.org/en-US/docs/Web/API/Request/headers)).
* request-no-cors: guard for a headers object obtained from a request created with [Request.mode](https://developer.mozilla.org/en-US/docs/Web/API/Request/mode) no-cors.
* response: guard for a headers object obtained from a response ([Response.headers](https://developer.mozilla.org/en-US/docs/Web/API/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**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#response_objects)

As you have seen above, [Response](https://developer.mozilla.org/en-US/docs/Web/API/Response) instances are returned when fetch() promises are resolved.

The most common response properties you'll use are:

* [Response.status](https://developer.mozilla.org/en-US/docs/Web/API/Response/status) — An integer (default value 200) containing the response status code.
* [Response.statusText](https://developer.mozilla.org/en-US/docs/Web/API/Response/statusText) — A string (default value ""), which corresponds to the HTTP status code message. Note that HTTP/2 [does not support](https://fetch.spec.whatwg.org/#concept-response-status-message) status messages.
* [Response.ok](https://developer.mozilla.org/en-US/docs/Web/API/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](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API), when you are providing a custom response to a received request using a [respondWith()](https://developer.mozilla.org/en-US/docs/Web/API/FetchEvent/respondWith) method:

JSCopy to Clipboard

const myBody = new Blob();

addEventListener("fetch", (event) => {

// ServiceWorker intercepting a fetch

event.respondWith(

new Response(myBody, {

headers: { "Content-Type": "text/plain" },

}),

);

});

The [Response()](https://developer.mozilla.org/en-US/docs/Web/API/Response/Response) constructor takes two optional arguments — a body for the response, and an init object (similar to the one that [Request()](https://developer.mozilla.org/en-US/docs/Web/API/Request/Request) accepts.)

**Note:** The static method [error()](https://developer.mozilla.org/en-US/docs/Web/API/Response/error_static) returns an error response. Similarly, [redirect()](https://developer.mozilla.org/en-US/docs/Web/API/Response/redirect_static) returns a response resulting in a redirect to a specified URL. These are also only relevant to Service Workers.

[**Body**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#body)

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

* [ArrayBuffer](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer)
* [TypedArray](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypedArray) (Uint8Array and friends)
* [DataView](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DataView)
* [Blob](https://developer.mozilla.org/en-US/docs/Web/API/Blob)
* [File](https://developer.mozilla.org/en-US/docs/Web/API/File)
* [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String), or a string literal
* [URLSearchParams](https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams)
* [FormData](https://developer.mozilla.org/en-US/docs/Web/API/FormData)

The [Request](https://developer.mozilla.org/en-US/docs/Web/API/Request) and [Response](https://developer.mozilla.org/en-US/docs/Web/API/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()](https://developer.mozilla.org/en-US/docs/Web/API/Request/arrayBuffer) / [Response.arrayBuffer()](https://developer.mozilla.org/en-US/docs/Web/API/Response/arrayBuffer)
* [Request.blob()](https://developer.mozilla.org/en-US/docs/Web/API/Request/blob) / [Response.blob()](https://developer.mozilla.org/en-US/docs/Web/API/Response/blob)
* [Request.formData()](https://developer.mozilla.org/en-US/docs/Web/API/Request/formData) / [Response.formData()](https://developer.mozilla.org/en-US/docs/Web/API/Response/formData)
* [Request.json()](https://developer.mozilla.org/en-US/docs/Web/API/Request/json) / [Response.json()](https://developer.mozilla.org/en-US/docs/Web/API/Response/json)
* [Request.text()](https://developer.mozilla.org/en-US/docs/Web/API/Request/text) / [Response.text()](https://developer.mozilla.org/en-US/docs/Web/API/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:

JSCopy to Clipboard

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 [options](https://developer.mozilla.org/en-US/docs/Web/API/fetch#options) parameter.

[**Feature detection**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#feature_detection)

Fetch API support can be detected by checking for the existence of [Headers](https://developer.mozilla.org/en-US/docs/Web/API/Headers), [Request](https://developer.mozilla.org/en-US/docs/Web/API/Request), [Response](https://developer.mozilla.org/en-US/docs/Web/API/Response) or [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch) on the [Window](https://developer.mozilla.org/en-US/docs/Web/API/Window) or [Worker](https://developer.mozilla.org/en-US/docs/Web/API/Worker) scope. For example:

JSCopy to Clipboard

if (window.fetch) {

// run my fetch request here

} else {

// do something with XMLHttpRequest?

}

[**Differences from jQuery.ajax()**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#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](https://developer.mozilla.org/en-US/docs/Web/API/Response/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](https://developer.mozilla.org/en-US/docs/Web/API/fetch#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.