FORM VALIDATION:

* [required](https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/required): Specifies whether a form field needs to be filled in before the form can be submitted.
* [minlength](https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/minlength) and [maxlength](https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/maxlength): Specifies the minimum and maximum length of textual data (strings)
* [min](https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/min) and [max](https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/max): Specifies the minimum and maximum values of numerical input types
* type: Specifies whether the data needs to be a number, an email address, or some other specific preset type.
* [pattern](https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/pattern): Specifies a [regular expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions) that defines a pattern the entered data needs to follow.

When an element is valid, the following things are true:

* The element matches the [:valid](https://developer.mozilla.org/en-US/docs/Web/CSS/:valid) CSS pseudo-class, which lets you apply a specific style to valid elements.
* If the user tries to send the data, the browser will submit the form, provided there is nothing else stopping it from doing so (e.g., JavaScript).

When an element is invalid, the following things are true:

* The element matches the [:invalid](https://developer.mozilla.org/en-US/docs/Web/CSS/:invalid) CSS pseudo-class, and sometimes other UI pseudo-classes (e.g., [:out-of-range](https://developer.mozilla.org/en-US/docs/Web/CSS/:out-of-range)) depending on the error, which lets you apply a specific style to invalid elements.
* If the user tries to send the data, the browser will block the form and display an error message.

**Note:** There are several errors that will prevent the form from being submitted, including a [badInput](https://developer.mozilla.org/en-US/docs/Web/API/ValidityState/badInput), [patternMismatch](https://developer.mozilla.org/en-US/docs/Web/API/ValidityState/patternMismatch), [rangeOverflow](https://developer.mozilla.org/en-US/docs/Web/API/ValidityState/rangeOverflow) or [rangeUnderflow](https://developer.mozilla.org/en-US/docs/Web/API/ValidityState/rangeUnderflow), [stepMismatch](https://developer.mozilla.org/en-US/docs/Web/API/ValidityState/stepMismatch), [tooLong](https://developer.mozilla.org/en-US/docs/Web/API/ValidityState/tooLong) or [tooShort](https://developer.mozilla.org/en-US/docs/Web/API/ValidityState/tooShort), [typeMismatch](https://developer.mozilla.org/en-US/docs/Web/API/ValidityState/typeMismatch), valueMissing, or a customError.

input:invalid:required {

background-image: linear-gradient(to right, pink, lightgreen);

}

Try submitting the form without a value. Note how the invalid input gets focus, a default error message ("Please fill out this field") appears, and the form is prevented from being sent.

The presence of the required attribute on any element that supports this attribute means the element matches the [:required](https://developer.mozilla.org/en-US/docs/Web/CSS/:required) pseudoclass whether it has a value or not. If the [<input>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input) has no value, the input will match the [:invalid](https://developer.mozilla.org/en-US/docs/Web/CSS/:invalid) pseudoclass.

**Note:** For good user experience, indicate to the user when form fields are required. It isn't only good user experience, it is required by WCAG [accessibility](https://developer.mozilla.org/en-US/docs/Learn/Accessibility) guidelines. Also, only require users to input data you actually need

* a — Matches one character that is a (not b, not aa, and so on).
* abc — Matches a, followed by b, followed by c.
* ab?c—Matches a, optionally followed by a single b, followed by c. ( ac or abc)
* ab\*c—Matches a, optionally followed by any number of bs, followed by c. ( ac , abc, abbbbbc, and so on).
* a|b — Matches one character that is a or b.
* abc|xyz — Matches exactly abc or exactly xyz (but not abcxyz or a or y, and so on).

<form>

<label for="choose">Would you prefer a banana or a cherry?</label>

<input id="choose" name="i\_like" required pattern="[Bb]anana|[Cc]herry">

<button>Submit</button>

</form>

In this example, the [<input>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input) element accepts one of four possible values: the strings "banana", "Banana", "cherry", or "Cherry". Regular expressions are case-sensitive, but we've made it support capitalized as well as lower-case versions using an extra "Aa" pattern nested inside square brackets.

If a non-empty value of the [<input>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input) doesn't match the regular expression's pattern, the input will match the [:invalid](https://developer.mozilla.org/en-US/docs/Web/CSS/:invalid) pseudoclass.

Texto

Descripción generada automáticamente

Browsers often don't let the user type a longer value than expected into text fields. A better user experience than just using maxlength is to also provide character count feedback in an accessible manner and let them edit their content down to size. An example of this is the character limit seen on Twitter when Tweeting. JavaScript, including [solutions using maxlength](https://github.com/mimo84/bootstrap-maxlength), can be used to provide this.

Texto

Descripción generada automáticamente

### [The Constraint Validation API](https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation#the_constraint_validation_api)

## [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.

// 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

## [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.

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

credentials: 'include'

});

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

Captura de pantalla con la imagen de una pantalla

Descripción generada automáticamente

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

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.

Texto

Descripción generada automáticamente

## [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" \o "FormData()) and [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch).

Texto

Descripción generada automáticamente

## [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" \o "FormData()) and [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch).

Texto

Descripción generada automáticamente

## [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).

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 re = /\n|\r|\r\n/gm;

let startIndex = 0;

let result;

for (;;) {

let result = re.exec(chunk);

if (!result) {

if (readerDone) {

break;

}

let remainder = chunk.substr(startIndex);

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

chunk = remainder + (chunk ? utf8Decoder.decode(chunk) : '');

startIndex = re.lastIndex = 0;

continue;

}

yield chunk.substring(startIndex, result.index);

startIndex = re.lastIndex;

}

if (startIndex < chunk.length) {

// last line didn't end in a newline char

yield chunk.substr(startIndex);

}

}

async function run() {

for await (let 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:

fetch('flowers.jpg')

.then(response => {

if (!response.ok) {

throw new Error('Network response was not OK');

}

return response.blob();

})

.then(myBlob => {

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:

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);

**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)

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

The contents can be queried and retrieved:

Texto

Descripción generada automáticamente

Some of these operations are only useful in [ServiceWorkers](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API" \o "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](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#guard)). Otherwise, they fail silently. For example:

Texto

Descripción generada automáticamente

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

Texto

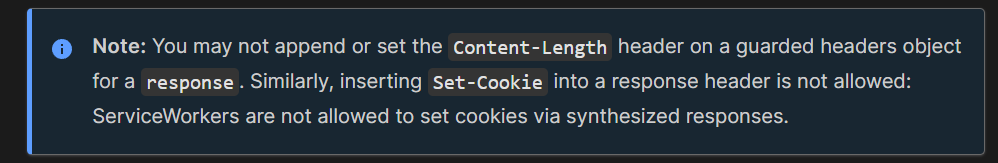
Descripción generada automáticamente

### [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.



## [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/" \l "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" \o "ServiceWorkers), 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" \o "respondWith()) method:

Texto

Descripción generada automáticamente

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.)

Texto

Descripción generada automáticamente

## [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:

Texto

Descripción generada automáticamente

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](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:

Texto

Descripción generada automáticamente