**What is form validation?**

Go to any popular site with a registration form, and you will notice that they provide feedback when you don't enter your data in the format they are expecting. You'll get messages such as:

* "This field is required" (You can't leave this field blank.)
* "Please enter your phone number in the format xxx-xxxx" (The form enforces three numbers followed by a dash, followed by four numbers.)
* "Please enter a valid email address" (Used if your entry is not in the format of "somebody@example.com.")
* "Your password needs to be between 8 and 30 characters long and contain one uppercase letter, one symbol, and a number."

This is called **form validation**. When you enter data, the web application checks it to see that the data is in the correct format. If the information is correctly formatted, the application allows the data to be submitted to the server and (usually) saved in a database; if the information isn't correctly formatted, it gives you an error message explaining what needs to be corrected. Form validation can be implemented in a number of different ways.

### **Different types of form validation**

There are two different types of form validation that you'll encounter on the web:

* **Client-side validation** is validation that occurs in the browser before the data has been submitted to the server. Client-side validation is more user-friendly than server-side validation because it gives an instant response. Client-side validation is further subdivided into the following categories:
  + **JavaScript** validation is coded using JavaScript. This validation is completely customizable.
  + **Built-in form validation** uses HTML5 form validation features. This validation generally doesn't require JavaScript. Built-in form validation has better performance than JavaScript. And, while highly customizable, native validation is not as customizable as JavaScript.
* **Server-side validation** is validation that occurs on the server after the data has been submitted. Server-side code is used to validate the data before the data is saved in the database or otherwise used by the application. If the data fails validation, a response is sent back to the client with corrections that the user needs to make. Server-side validation is your application's last line of defense against incorrect or malicious data. All popular [server-side frameworks](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks) have features for **validating** and **sanitizing** data, or making it safe.

There are two different types of form validation that you'll encounter on the web:

* **Client-side validation** is validation that occurs in the browser before the data has been submitted to the server. Client-side validation is more user-friendly than server-side validation because it gives an instant response. Client-side validation is further subdivided into the following categories:
  + **JavaScript** validation is coded using JavaScript. This validation is completely customizable.
  + **Built-in form validation** uses HTML5 form validation features. This validation generally doesn't require JavaScript. Built-in form validation has better performance than JavaScript. And, while highly customizable, native validation is not as customizable as JavaScript.
* **Server-side validation** is validation that occurs on the server after the data has been submitted. Server-side code is used to validate the data before the data is saved in the database or otherwise used by the application. If the data fails validation, a response is sent back to the client with corrections that the user needs to make. Server-side validation is your application's last line of defense against incorrect or malicious data. All popular [server-side frameworks](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks) have features for **validating** and **sanitizing** data, or making it safe.

For good user experience and security, developers should use a combination of both client-side and server-side validation. Client-side validation is more user-friendly than server-side validation because it provides errors during form completion rather than requiring the entire form to be completed, submitted, analyzed, and returned. Server-side validation should be considered required because client-side error checking can be bypassed and network requests can be altered.

## Using built-in form validation

One of the features of [HTML5](https://developer.mozilla.org/en-US/docs/HTML/HTML5) form controls is the ability to validate most user data without relying on scripts. This is done by using [validation attributes](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/Constraint_validation#Validation-related_attributes) on form elements. Validation attributes allow you to specify rules for a form input, such as whether a value must be filled in; the minimum and maximum length of the data; the minimum and maximum values; whether the data needs to be a number, an email address, or something else; and a regular expression pattern that the data must match. In supporting browsers, if the entered data follows all of the specified rules, it is considered valid; if not, it is considered invalid.

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, 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.

### **Validation constraints on input elements — starting simple**

In this section, we'll look at some of the different form control attributes that can be used to validate [<input>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input) elements.

Let's start with a simple example: an input that allows you to choose whether you prefer a banana or a cherry. This example involves a simple text [<input>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input) with a matching label and a submit [<button>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/button). Find the source code on GitHub at [fruit-start.html](https://github.com/mdn/learning-area/blob/master/html/forms/form-validation/fruit-start.html) and a live example below.

Open in CodePenOpen in JSFiddle

To begin, make a copy of fruit-start.html in a new directory on your hard drive.

### **The required attribute**

The simplest HTML5 validation feature is the [required](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#attr-required) attribute. To make an input mandatory, mark the element with this attribute. When this attribute is set, the form won't submit and will display an error message when the input is empty. The input will also be considered invalid.

Add a required attribute to your input, as shown below.

<form>

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

<input id="choose" name="i\_like" required>

<button>Submit</button>

</form>

Note the CSS that is included in the example file:

input:invalid {

border: 2px dashed red;

}

input:valid {

border: 2px solid black;

}

### **Customized error messages**

As you saw in the examples above, each time a user tries to submit an invalid form, the browser displays an error message. The way this message is displayed depends on the browser.

These automated messages have two drawbacks:

* There is no standard way to change their look and feel with CSS.
* They depend on the browser locale, which means that you can have a page in one language but an error message displayed in another language.

| **French versions of feedback messages on an English page** | |
| --- | --- |
| **Browser** | **Rendering** |
| Firefox 17 (Windows 7) | Example of an error message with Firefox in French on an English page |
| Chrome 22 (Windows 7) | Example of an error message with Chrome in French on an English page |
| Opera 12.10 (Mac OSX) | Example of an error message with Opera in French on an English page |

To customize the appearance and text of these messages, you must use JavaScript; there is no way to do it using only HTML and CSS.

HTML5 provides the [constraint validation API](https://developer.mozilla.org/en-US/docs/Web/API/Constraint_validation) to check and customize the state of a form element. Among other things, it's possible to change the text of the error message. Here's a quick example:

<form>

<label for="mail">I would like you to provide me an e-mail</label>

<input type="email" id="mail" name="mail">

<button>Submit</button>

</form>

In JavaScript, you call the [setCustomValidity()](https://developer.mozilla.org/en-US/docs/HTML/HTML5/Constraint_validation" \l "Constraint_API's_element.setCustomValidity()) method:

var email = document.getElementById("mail");

email.addEventListener("input", function (event) {

if (email.validity.typeMismatch) {

email.setCustomValidity("I expect an e-mail, darling!");

} else {

email.setCustomValidity("");

}

});

Open in CodePenOpen in JSFiddle

Note: if you set a custom validity message you must set the custom validity message to the empty string when the value is valid or the form will never be submitted.

**FETCH :**

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 HTTP pipeline, such as requests and responses. It also provides a global [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/GlobalFetch/fetch) method that provides an easy, logical way to fetch resources asynchronously across the network.

This kind of functionality was previously achieved using [XMLHttpRequest](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest" \o "Use XMLHttpRequest (XHR) objects to interact with servers. You can retrieve data from a URL without having to do a full page refresh. This enables a Web page to update just part of a page without disrupting what the user is doing.). Fetch provides a better alternative that can be easily used by other technologies such as [Service Workers](https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorker_API). Fetch also provides a single logical place to define other HTTP-related concepts such as CORS and extensions to HTTP.

The fetch specification differs from jQuery.ajax() in three main ways:

* The Promise returned from fetch() **won’t reject on HTTP error status** even if the response is an HTTP 404 or 500. Instead, it will resolve normally (with ok status set to false), and it will only reject on network failure or if anything prevented the request from completing.
* fetch() **won't receive cross-site cookies;**you can’t establish a cross site session using fetch. [Set-Cookie](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie) headers from other sites are silently ignored.
* fetch **won’t send cookies**, unless you set the *credentials* [init option](https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch" \l "Parameters). (Since [Aug 25, 2017](https://github.com/whatwg/fetch/pull/585). The spec changed the default credentials policy to same-origin. Firefox changed since 61.0b13.)

A basic fetch request is really simple to set up. Have a look at the following code:

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

const myJson = await response.json();

console.log(JSON.stringify(myJson));

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 returns a promise containing the response (a [Response](https://developer.mozilla.org/en-US/docs/Web/API/Response) object).

This is just an HTTP response, not the actual JSON. To extract the JSON body content from the response, we use the [json()](https://developer.mozilla.org/en-US/docs/Web/API/Body/json) method (defined on the [Body](https://developer.mozilla.org/en-US/docs/Web/API/Body) mixin, which is implemented by both the [Request](https://developer.mozilla.org/en-US/docs/Web/API/Request) and [Response](https://developer.mozilla.org/en-US/docs/Web/API/Response) objects.)

**Note**: The Body mixin also has similar methods to extract other types of body content; see the [Body](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#Body) section for more.

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/GlobalFetch/fetch) for the full options available, and more details.

// Example POST method implementation:

try {

const data = await postData('http://example.com/answer', { answer: 42 });

console.log(JSON.stringify(data)); // JSON-string from `response.json()` call

} catch (error) {

console.error(error);

}

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

referrer: 'no-referrer', // no-referrer, \*client

body: JSON.stringify(data) // body data type must match "Content-Type" header

});

return await response.json(); // parses JSON response into native JavaScript objects

}

### **Sending a request with credentials included**

To cause browsers to send a request with credentials included, even for a cross-origin call, add credentials: 'include' to the init the 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'.

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

const url = 'https://example.com/profile';

const data = { username: 'example' };

try {

const response = await fetch(url, {

method: 'POST', // or 'PUT'

body: JSON.stringify(data), // data can be `string` or {object}!

headers: {

'Content-Type': 'application/json'

}

});

const json = await response.json();

console.log('Success:', JSON.stringify(json));

} catch (error) {

console.error('Error:', error);

}

### **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 "The FormData() constructor creates a new FormData object.) and [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch).

const formData = new FormData();

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

formData.append('username', 'abc123');

formData.append('avatar', fileField.files[0]);

try {

const response = await fetch('https://example.com/profile/avatar', {

method: 'PUT',

body: formData

});

const result = await response.json();

console.log('Success:', JSON.stringify(result));

} catch (error) {

console.error('Error:', error);

}

### **Checking that the fetch was successful**

A [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/GlobalFetch/fetch) promise will reject with a [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError" \o "The TypeError object represents an error when an operation could not be performed, typically (but not exclusively) when a value is not of the expected type.) 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" \o "The ok read-only property of the Response interface contains a Boolean stating whether the response was successful (status in the range 200-299) or not.) property has a value of true. The code would look something like this:

try {

const response = await fetch('flowers.jpg');

if (!response.ok) {

throw new Error('Network response was not ok.');

}

const myBlob = await response.blob();

const objectURL = URL.createObjectURL(myBlob);

myImage.src = objectURL;

} catch (error) {

console.log('There has been a problem with your fetch operation: ', error.message);

}

### **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" \o "The headers read-only property of the Request interface contains the Headers object associated with the request.)).
* 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" \o "The mode read-only property of the Request interface contains the mode of the request (e.g., cors, no-cors, same-origin, or navigate.) This is used to determine if cross-origin requests lead to valid responses, and which properties of the response are readable.) no-cors.
* response: guard for a Headers obtained from a response ([Response.headers](https://developer.mozilla.org/en-US/docs/Web/API/Response/headers" \o "The headers read-only property of the Response interface contains the Headers object associated with the response.)).
* immutable: Mostly used for ServiceWorkers; renders a headers object read-only.

**Note**: You may not append or set a request guarded Headers’ Content-Length header. 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](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 "OK"), which corresponds to the HTTP status code message.
* [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](https://developer.mozilla.org/en-US/docs/Web/API/Boolean).

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/ServiceWorker_API" \o "The documentation about this has not yet been written; please consider contributing!), 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 "The respondWith() method of FetchEvent prevents the browser's default fetch handling, and allows you to provide a promise for a Response yourself.) method:

const myBody = new Blob();

addEventListener('fetch', function(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) simply returns an error response. Similarly, [redirect()](https://developer.mozilla.org/en-US/docs/Web/API/Response/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](https://developer.mozilla.org/en-US/docs/Web/API/ArrayBuffer)
* [ArrayBufferView](https://developer.mozilla.org/en-US/docs/Web/API/ArrayBufferView) (Uint8Array and friends)
* [Blob](https://developer.mozilla.org/en-US/docs/Web/API/Blob)/File
* string
* [URLSearchParams](https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams)
* [FormData](https://developer.mozilla.org/en-US/docs/Web/API/FormData)

The [Body](https://developer.mozilla.org/en-US/docs/Web/API/Body) mixin defines the following methods to extract a body (implemented by both [Request](https://developer.mozilla.org/en-US/docs/Web/API/Request) and [Response](https://developer.mozilla.org/en-US/docs/Web/API/Response)). These all return a promise that is eventually resolved with the actual content.

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

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.