/// mdn web docs _

Client-side form validation

Before submitting data to the server, it is important to ensure all required form controls are filled out, in the correct format. This is called **client-side form validation**, and helps ensure data submitted matches the requirements set forth in the various form controls. This article leads you through basic concepts and examples of client-side form validation.

Prerequisites:	Computer literacy, a reasonable understanding of [HTML](#), [CSS](#), and [JavaScript](#).
Objective:	To understand what client-side form validation is, why it's important, and how to apply various techniques to implement it.

Client-side validation is an initial check and an important feature of good user experience; by catching invalid data on the client-side, the user can fix it straight away. If it gets to the server and is then rejected, a noticeable delay is caused by a round trip to the server and then back to the client-side to tell the user to fix their data.

However, client-side validation *should not be considered* an exhaustive security measure! Your apps should always perform security checks on any form-submitted data on the *server-side* **as well** as the client-side, because client-side validation is too easy to bypass, so malicious users can still easily send bad data through to your server. Read [Website security](#) for an idea of what *could* happen; implementing server-side validation is somewhat beyond the scope of this module, but you should bear it in mind.

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" (A specific data format is required for it to be considered valid).

- "Please enter a valid email address" (the data you entered is not in the right format).

- "Your password needs to be between 8 and 30 characters long and contain one uppercase letter, one symbol, and a number." (A very specific data format is required for your data).

This is called **form validation**. When you enter data, the browser and/or the web server will check to see that the data is in the correct format and within the constraints set by the application. Validation done in the browser is called **client-side** validation, while validation done on the server is called **server-side** validation. In this chapter we are focusing on client-side validation.

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 the user an error message explaining what needs to be corrected, and lets them try again.

We want to make filling out web forms as easy as possible. So why do we insist on validating our forms? There are three main reasons:

- **We want to get the right data, in the right format.** Our applications won't work properly if our users' data is stored in the wrong format, is incorrect, or is omitted altogether.

- **We want to protect our users' data**. Forcing our users to enter secure passwords makes it easier to protect their account information.

- **We want to protect ourselves**. There are many ways that malicious users can misuse unprotected forms to damage the application. See [Website security](#).

> ⚠️ **Warning:** Never trust data passed to your server from the client. Even if your form is validating correctly and preventing malformed input on the client-side, a malicious user can still alter the network request.

Different types of client-side validation

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

- **Built-in form validation** uses HTML form validation features, which we've discussed in many places throughout this module. This validation generally doesn't require much JavaScript. Built-in form validation has better performance than JavaScript, but it is not as customizable as JavaScript validation.

- **JavaScript** validation is coded using JavaScript. This validation is completely customizable, but you need to create it all (or use a library).

Using built-in form validation

One of the most significant features of [modern form controls](#) is the ability to validate most user data without relying on JavaScript. This is done by using validation attributes on form elements. We've seen many of these earlier in the course, but to recap:

- `required` : Specifies whether a form field needs to be filled in before the form can be submitted.

- `minlength` and `maxlength` : Specifies the minimum and maximum length of textual data (strings).

- `min` and `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` : Specifies a [regular expression](#) that defines a pattern the entered data needs to follow.

If the data entered in a form field follows all of the rules specified by the above attributes, 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` 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` CSS pseudo-class, and sometimes other UI pseudo-classes (e.g., `: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.

> ℹ️ **Note:** There are several errors that will prevent the form from being submitted, including a `badInput` , `patternMismatch` , `rangeOverflow` or `rangeUnderflow` , `stepMismatch` , `tooLong` or `tooShort` , `typeMismatch` , `valueMissing` , or a `customError` .

Built-in form validation examples

In this section, we'll test out some of the attributes that we discussed above.

Simple start file

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>` with an associated `<label>` and a submit `<button>`. Find the source code on GitHub at fruit-start.html ⧉ and a live example below.

HTML	</> Play

```html
<form>
  <label for="choose">Would you prefer a banana or cherry?</label>
  <input id="choose" name="i-like" />
  <button>Submit</button>
</form>
```

CSS	</> Play

```css
input:invalid {
  border: 2px dashed red;
}

input:valid {
  border: 2px solid black;
}
```

	</> Play

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

The required attribute

The simplest HTML validation feature is the `required` attribute. To make an input mandatory, add this attribute to the element. When this attribute is set, the element matches the `:required` UI pseudo-class and the form won't submit, displaying an

error message on submission when the input is empty. While empty, the input will also be considered invalid, matching the `:invalid` UI pseudo-class.

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

HTML	</> Play

```html
<form>
  <label for="choose">Would you prefer a banana or cherry? (required)</label>
  <input id="choose" name="i-like" required />
  <button>Submit</button>
</form>
```

Note the CSS that is included in the example file:

CSS	</> Play

```css
input:invalid {
  border: 2px dashed red;
}

input:invalid:required {
  background-image: linear-gradient(to right, pink, lightgreen);
}

input:valid {
  border: 2px solid black;
}
```

This CSS causes the input to have a red dashed border when it is invalid and a more subtle solid black border when valid. We also added a background gradient when the input is required *and* invalid. Try out the new behavior in the example below:

	</> Play

> ℹ️ **Note:** You can find this example live on GitHub as <u>fruit-validation.html</u> ⤴. See also the <u>source code</u> ⤴.

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` pseudo-class whether it has a value or not. If the `<input>` has no value, the `input` will match the `:invalid` pseudo-class.

> ℹ️ **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 <u>accessibility</u> guidelines. Also, only require users to input data you actually need: For example, why do you really need to know someone's gender or title?

Validating against a regular expression

Another useful validation feature is the `pattern` attribute, which expects a [Regular Expression](#) as its value. A regular expression (regexp) is a pattern that can be used to match character combinations in text strings, so regexps are ideal for form validation and serve a variety of other uses in JavaScript.

Regexps are quite complex, and we don't intend to teach you them exhaustively in this article. Below are some examples to give you a basic idea of how they work.

- `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 `b` s, 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).

There are many more possibilities that we don't cover here. For a complete list and
many examples, consult our [Regular expression](#) documentation.

Let's implement an example. Update your HTML to add a `pattern` attribute like this:

```
HTML                                                        </> Play   ⧉

<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>
```

This gives us the following update — try it out:

```
                                                           </> Play

```

> ℹ **Note:** You can find this example live on GitHub as [fruit-pattern.html](#) ⧉ (see
> also the [source code](#) ⧉.)

In this example, the `<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.

At this point, try changing the value inside the `pattern` attribute to equal some of the examples you saw earlier, and look at how that affects the values you can enter to make the input value valid. Try writing some of your own, and see how it goes. Make them fruit-related where possible so that your examples make sense!

If a non-empty value of the `<input>` doesn't match the regular expression's pattern, the `input` will match the `:invalid` pseudo-class.

> **ⓘ** **Note:** Some `<input>` element types don't need a `pattern` attribute to be validated against a regular expression. Specifying the `email` type, for example, validates the inputs value against a well-formed email address pattern or a pattern matching a comma-separated list of email addresses if it has the `multiple` attribute.

> **ⓘ** **Note:** The `<textarea>` element doesn't support the `pattern` attribute.

Constraining the length of your entries

You can constrain the character length of all text fields created by `<input>` or `<textarea>` by using the `minlength` and `maxlength` attributes. A field is invalid if it has a value and that value has fewer characters than the `minlength` value or more than the `maxlength` value.

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 when posting on social media. JavaScript, including solutions using `maxlength` ⬈, can be used to provide this.

> **ⓘ** **Note:** Length constraints are never reported if the value is set programmatically. They are only reported for user-provided input.

Constraining the values of your entries

For number fields (i.e. `<input type="number">`), the `min` and `max` attributes can be used to provide a range of valid values. If the field contains a value outside this range, it will be invalid.

Let's look at another example. Create a new copy of the [fruit-start.html](#) ⬈ file.

Now delete the contents of the `<body>` element, and replace it with the following:

HTML   </> Play   ⬚

```html
<form>
  <div>
    <label for="choose">Would you prefer a banana or a cherry?</label>
    <input
      type="text"
      id="choose"
      name="i-like"
      required
      minlength="6"
      maxlength="6" />
  </div>
  <div>
    <label for="number">How many would you like?</label>
    <input type="number" id="number" name="amount" value="1" min="1" max="10" />
  </div>
  <div>
    <button>Submit</button>
  </div>
</form>
```

- Here you'll see that we've given the `text` field a `minlength` and `maxlength` of six, which is the same length as banana and cherry.

- We've also given the `number` field a `min` of one and a `max` of ten. Entered numbers outside this range will show as invalid; users won't be able to use the increment/decrement arrows to move the value outside of this range. If the user

manually enters a number outside of this range, the data is invalid. The number is not required, so removing the value will still result in a valid value.

Here is the example running live:

> </> Play

> **Note:** You can find this example live on GitHub as <u>fruit-length.html</u> ⧉. See also the <u>source code</u> ⧉.

> **Note:** `<input type="number">` (and other types, such as `range` and `date`) can also take a `step` attribute, which specifies what increment the value will go up or down by when the input controls are used (such as the up and down number buttons). In the above example we've not included a `step` attribute, so the value defaults to `1`. This means that floats, like 3.2, will also show as invalid.

Full example

Here is a full example to show usage of HTML's built-in validation features. First, some HTML:

HTML   </> Play

```html
<form>
  <fieldset>
    <legend>
      Do you have a driver's license?<span aria-label="required">*</span>
    </legend>
```

```html
    <!-- While only one radio button in a same-named group can be selected at a time,
         and therefore only one radio button in a same-named group having the
"required"
         attribute suffices in making a selection a requirement -->
    <input type="radio" required name="driver" id="r1" value="yes" /><label
      for="r1"
      >Yes</label
    >
    <input type="radio" required name="driver" id="r2" value="no" /><label
      for="r2"
      >No</label
    >
  </fieldset>
  <p>
    <label for="n1">How old are you?</label>
    <!-- The pattern attribute can act as a fallback for browsers which
         don't implement the number input type but support the pattern attribute.
         Please note that browsers that support the pattern attribute will make it
         fail silently when used with a number field.
         Its usage here acts only as a fallback -->
    <input
      type="number"
      min="12"
      max="120"
      step="1"
      id="n1"
      name="age"
      pattern="\d+" />
  </p>
  <p>
    <label for="t1"
      >What's your favorite fruit?<span aria-label="required">*</span></label
    >
    <input
      type="text"
      id="t1"
      name="fruit"
      list="l1"
      required
      pattern="[Bb]anana|[Cc]herry|[Aa]pple|[Ss]trawberry|[Ll]emon|[Oo]range" />
    <datalist id="l1">
      <option>Banana</option>
```

```html
        <option>Cherry</option>
        <option>Apple</option>
        <option>Strawberry</option>
        <option>Lemon</option>
        <option>Orange</option>
      </datalist>
    </p>
    <p>
      <label for="t2">What's your email address?</label>
      <input type="email" id="t2" name="email" />
    </p>
    <p>
      <label for="t3">Leave a short message</label>
      <textarea id="t3" name="msg" maxlength="140" rows="5"></textarea>
    </p>
    <p>
      <button>Submit</button>
    </p>
</form>
```

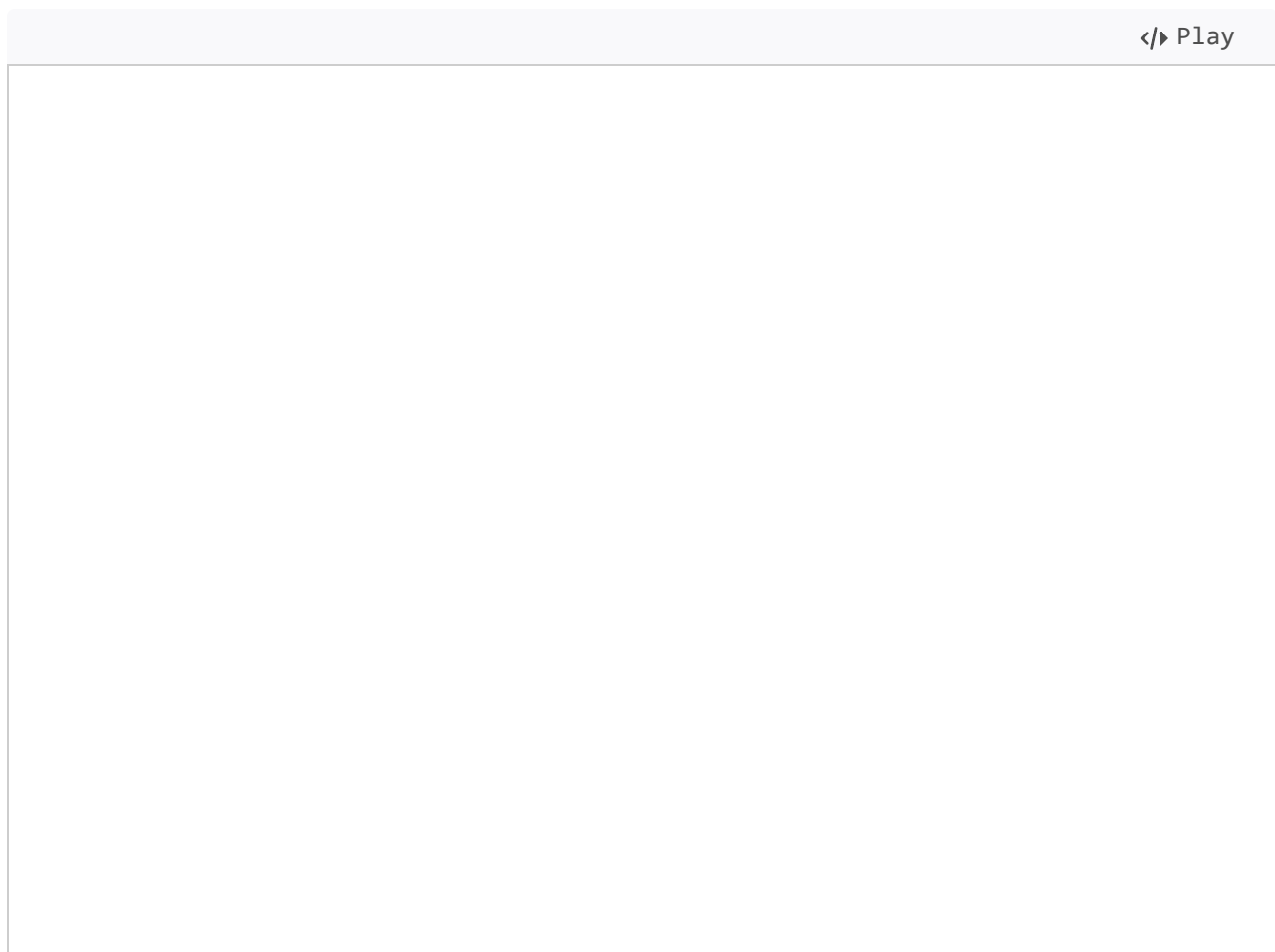And now some CSS to style the HTML:

```
CSS                                                    </> Play  

form {
  font: 1em sans-serif;
  max-width: 320px;
}


p > label {
  display: block;
}


input[type="text"],
input[type="email"],
input[type="number"],
textarea,
fieldset {
  width: 100%;
  border: 1px solid #333;
  box-sizing: border-box;
```

```
}

input:invalid {
  box-shadow: 0 0 5px 1px red;
}

input:focus:invalid {
  box-shadow: none;
}
```

This renders as follows:

See [Validation-related attributes](#) for a complete list of attributes that can be used to constrain input values and the input types that support them.

> ℹ️ **Note:** You can find this example live on GitHub as <u>full-example.html</u> ↗ (see also the <u>source code</u> ↗.)

Validating forms using JavaScript

You must use JavaScript if you want to take control over the look and feel of native error messages. In this section we will look at the different ways to do this.

The Constraint Validation API

The Constraint Validation API consists of a set of methods and properties available on the following form element DOM interfaces:

- <u>HTMLButtonElement</u> (represents a `<button>` element)
- <u>HTMLFieldSetElement</u> (represents a `<fieldset>` element)
- <u>HTMLInputElement</u> (represents an `<input>` element)
- <u>HTMLOutputElement</u> (represents an `<output>` element)
- <u>HTMLSelectElement</u> (represents a `<select>` element)
- <u>HTMLTextAreaElement</u> (represents a `<textarea>` element)

The Constraint Validation API makes the following properties available on the above elements.

- `validationMessage` : Returns a localized message describing the validation constraints that the control doesn't satisfy (if any). If the control is not a candidate for constraint validation (`willValidate` is `false`) or the element's value satisfies its constraints (is valid), this will return an empty string.

- `validity` : Returns a `ValidityState` object that contains several properties describing the validity state of the element. You can find full details of all the available properties in the `ValidityState` reference page; below is listed a few of the more common ones:

- `patternMismatch` : Returns `true` if the value does not match the specified `pattern` , and `false` if it does match. If true, the element matches the `:invalid` CSS pseudo-class.

- `tooLong` : Returns `true` if the value is longer than the maximum length specified by the `maxlength` attribute, or `false` if it is shorter than or equal to the maximum. If true, the element matches the `:invalid` CSS pseudo-class.

- `tooShort` : Returns `true` if the value is shorter than the minimum length specified by the `minlength` attribute, or `false` if it is greater than or equal to the minimum. If true, the element matches the `:invalid` CSS pseudo-class.

- `rangeOverflow` : Returns `true` if the value is greater than the maximum specified by the `max` attribute, or `false` if it is less than or equal to the maximum. If true, the element matches the `:invalid` and `:out-of-range` CSS pseudo-classes.

- `rangeUnderflow` : Returns `true` if the value is less than the minimum specified by the `min` attribute, or `false` if it is greater than or equal to the minimum. If true, the element matches the `:invalid` and `:out-of-range` CSS pseudo-classes.

- `typeMismatch` : Returns `true` if the value is not in the required syntax (when `type` is `email` or `url`), or `false` if the syntax is correct. If `true` , the element matches the `:invalid` CSS pseudo-class.

- `valid` : Returns `true` if the element meets all its validation constraints, and is therefore considered to be valid, or `false` if it fails any constraint. If true, the element matches the `:valid` CSS pseudo-class; the `:invalid` CSS pseudo-class otherwise.

- `valueMissing` : Returns `true` if the element has a `required` attribute, but no value, or `false` otherwise. If true, the element matches the `:invalid` CSS pseudo-class.

- `willValidate` : Returns `true` if the element will be validated when the form is submitted; `false` otherwise.

The Constraint Validation API also makes the following methods available on the above elements and the `form` element.

- `checkValidity()` : Returns `true` if the element's value has no validity problems; `false` otherwise. If the element is invalid, this method also fires an [invalid event](#) on the element.

- `reportValidity()` : Reports invalid field(s) using events. This method is useful in combination with `preventDefault()` in an `onSubmit` event handler.

- `setCustomValidity(message)` : Adds a custom error message to the element; if you set a custom error message, the element is considered to be invalid, and the specified error is displayed. This lets you use JavaScript code to establish a validation failure other than those offered by the standard HTML validation constraints. The message is shown to the user when reporting the problem.

Implementing a customized error message

As you saw in the HTML validation constraint examples earlier, 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, as seen in the following Firefox screenshot.
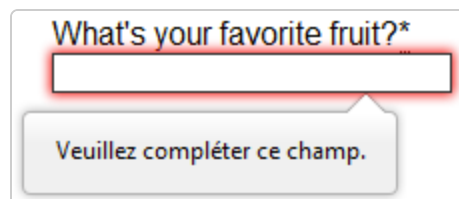
Customizing these error messages is one of the most common use cases of the Constraint Validation API. Let's work through a simple example of how to do this.

We'll start with some simple HTML (feel free to put this in a blank HTML file; use a fresh copy of [fruit-start.html](#) ⬀ as a basis, if you like):

```html
HTML

<form>
  <label for="mail">
    I would like you to provide me with an email address:
  </label>
  <input type="email" id="mail" name="mail" />
  <button>Submit</button>
</form>
```

And add the following JavaScript to the page:

```js
JS

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

email.addEventListener("input", (event) => {
  if (email.validity.typeMismatch) {
    email.setCustomValidity("I am expecting an email address!");
  } else {
    email.setCustomValidity("");
  }
});
```

Here we store a reference to the email input, then add an event listener to it that runs the contained code each time the value inside the input is changed.

Inside the contained code, we check whether the email input's `validity.typeMismatch` property returns `true`, meaning that the contained value doesn't match the pattern for a well-formed email address. If so, we call the `setCustomValidity()` method with a custom message. This renders the input invalid, so that when you try to submit the form, submission fails and the custom error message is displayed.

If the `validity.typeMismatch` property returns `false`, we call the `setCustomValidity()` method with an empty string. This renders the input valid, so the form will submit.

You can try it out below:

> ℹ **Note:** You can find this example live on GitHub as <u>custom-error-</u>
> <u>message.html</u> ☐ (see also the <u>source code</u> ☐.)

A more detailed example

Now that we've seen a really simple example, let's see how we can use this API to build some slightly more complex custom validation.

First, the HTML. Again, feel free to build this along with us:

```html
HTML

<form novalidate>
  <p>
    <label for="mail">
      <span>Please enter an email address:</span>
      <input type="email" id="mail" name="mail" required minlength="8" />
      <span class="error" aria-live="polite"></span>
    </label>
  </p>
  <button>Submit</button>
</form>
```

This simple form uses the `novalidate` attribute to turn off the browser's automatic validation; this lets our script take control over validation. However, this doesn't disable support for the constraint validation API nor the application of CSS pseudo-classes like `:valid`, etc. That means that even though the browser doesn't automatically check the validity of the form before sending its data, you can still do it yourself and style the form accordingly.

Our input to validate is an `<input type="email">`, which is `required`, and has a `minlength` of 8 characters. Let's check these using our own code, and show a custom error message for each one.

We are aiming to show the error messages inside a `<span>` element. The `aria-live` attribute is set on that `<span>` to make sure that our custom error message will be presented to everyone, including it being read out to screen reader users.

> ⓘ **Note:** A key point here is that setting the `novalidate` attribute on the form is what stops the form from showing its own error message bubbles, and allows us to instead display the custom error messages in the DOM in some manner of our own choosing.

Now onto some basic CSS to improve the look of the form slightly, and provide some visual feedback when the input data is invalid:

```css
body {
  font: 1em sans-serif;
  width: 200px;
  padding: 0;
  margin: 0 auto;
}

p * {
  display: block;
}

input[type="email"] {
  appearance: none;

  width: 100%;
  border: 1px solid #333;
  margin: 0;

  font-family: inherit;
  font-size: 90%;
```

```css
  box-sizing: border-box;
}

/* This is our style for the invalid fields */
input:invalid {
  border-color: #900;
  background-color: #fdd;
}

input:focus:invalid {
  outline: none;
}

/* This is the style of our error messages */
.error {
  width: 100%;
  padding: 0;

  font-size: 80%;
  color: white;
  background-color: #900;
  border-radius: 0 0 5px 5px;

  box-sizing: border-box;
}

.error.active {
  padding: 0.3em;
}
```

Now let's look at the JavaScript that implements the custom error validation.

```js
JS

// There are many ways to pick a DOM node; here we get the form itself and the email
// input box, as well as the span element into which we will place the error message.
const form = document.querySelector("form");
const email = document.getElementById("mail");
const emailError = document.querySelector("#mail + span.error");
```

```
email.addEventListener("input", (event) => {
  // Each time the user types something, we check if the
  // form fields are valid.

  if (email.validity.valid) {
    // In case there is an error message visible, if the field
    // is valid, we remove the error message.
    emailError.textContent = ""; // Reset the content of the message
    emailError.className = "error"; // Reset the visual state of the message
  } else {
    // If there is still an error, show the correct error
    showError();
  }
});

form.addEventListener("submit", (event) => {
  // if the email field is valid, we let the form submit
  if (!email.validity.valid) {
    // If it isn't, we display an appropriate error message
    showError();
    // Then we prevent the form from being sent by canceling the event
    event.preventDefault();
  }
});

function showError() {
  if (email.validity.valueMissing) {
    // If the field is empty,
    // display the following error message.
    emailError.textContent = "You need to enter an email address.";
  } else if (email.validity.typeMismatch) {
    // If the field doesn't contain an email address,
    // display the following error message.
    emailError.textContent = "Entered value needs to be an email address.";
  } else if (email.validity.tooShort) {
    // If the data is too short,
    // display the following error message.
    emailError.textContent = `Email should be at least ${email.minLength} characters;
you entered ${email.value.length}.`;
  }

  // Set the styling appropriately
```
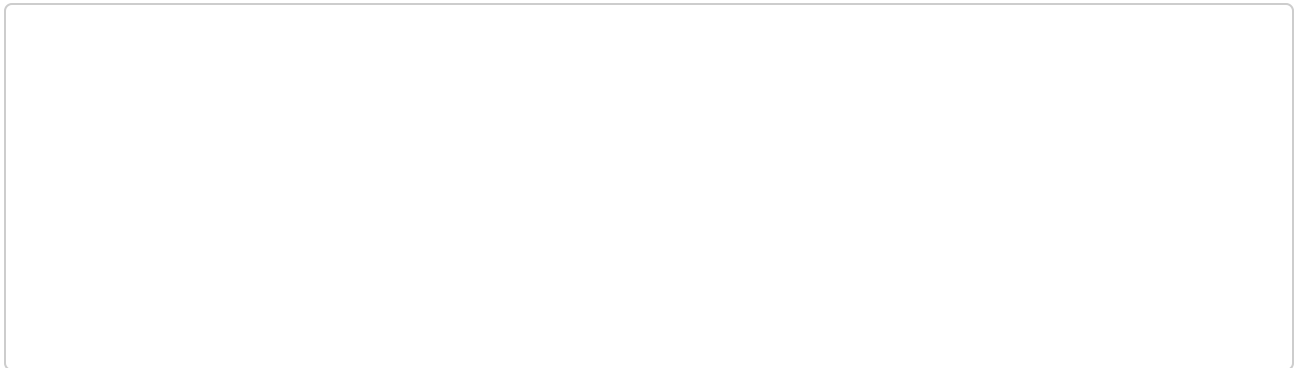
```
    emailError.className = "error active";
}
```

The comments explain things pretty well, but briefly:

- Every time we change the value of the input, we check to see if it contains valid data. If it has then we remove any error message being shown. If the data is not valid, we run `showError()` to show the appropriate error.

- Every time we try to submit the form, we again check to see if the data is valid. If so, we let the form submit. If not, we run `showError()` to show the appropriate error, and stop the form submitting with `preventDefault()`.

- The `showError()` function uses various properties of the input's `validity` object to determine what the error is, and then displays an error message as appropriate.

Here is the live result:

> ℹ️ **Note:** You can find this example live on GitHub as <u>detailed-custom-validation.html</u> ⧉. See also the <u>source code</u> ⧉.

The constraint validation API gives you a powerful tool to handle form validation, letting you have enormous control over the user interface above and beyond what you can do with HTML and CSS alone.

Validating forms without a built-in API

In some cases, such as custom controls, you won't be able to or won't want to use the Constraint Validation API. You're still able to use JavaScript to validate your form, but you'll just have to write your own.

To validate a form, ask yourself a few questions:

What kind of validation should I perform?

You need to determine how to validate your data: string operations, type conversion, regular expressions, and so on. It's up to you.

What should I do if the form doesn't validate?

This is clearly a UI matter. You have to decide how the form will behave. Does the form send the data anyway? Should you highlight the fields that are in error? Should you display error messages?

How can I help the user to correct invalid data?

In order to reduce the user's frustration, it's very important to provide as much helpful information as possible in order to guide them in correcting their inputs. You should offer up-front suggestions so they know what's expected, as well as clear error messages. If you want to dig into form validation UI requirements, here are some useful articles you should read:

- Help users enter the right data in forms 🗗

- Validating input 🗗

- How to Report Errors in Forms: 10 Design Guidelines 🗗

An example that doesn't use the constraint validation API

In order to illustrate this, the following is a simplified version of the previous example without the Constraint Validation API.

The HTML is almost the same; we just removed the HTML validation features.

HTML   </> Play   ⬚

```html
<form>
  <p>
    <label for="mail">
      <span>Please enter an email address:</span>
      <input type="text" id="mail" name="mail" />
      <span class="error" aria-live="polite"></span>
    </label>
  </p>
  <button>Submit</button>
</form>
```

Similarly, the CSS doesn't need to change very much; we've just turned the `:invalid` CSS pseudo-class into a real class and avoided using the attribute selector.

CSS   </> Play   ⬚

```css
body {
  font: 1em sans-serif;
  width: 200px;
  padding: 0;
  margin: 0 auto;
}

form {
  max-width: 200px;
}

p * {
  display: block;
}

input#mail {
  appearance: none;
  width: 100%;
  border: 1px solid #333;
  margin: 0;

  font-family: inherit;
  font-size: 90%;
```

```css
  box-sizing: border-box;
}

/* This is our style for the invalid fields */
input.invalid {
  border-color: #900;
  background-color: #fdd;
}

input:focus:invalid {
  outline: none;
}

/* This is the style of our error messages */
.error {
  width: 100%;
  padding: 0;

  font-size: 80%;
  color: white;
  background-color: #900;
  border-radius: 0 0 5px 5px;
  box-sizing: border-box;
}

.error.active {
  padding: 0.3em;
}
```

The big changes are in the JavaScript code, which needs to do much more heavy
lifting.

JS   </> Play

```js
const form = document.querySelector("form");
const email = document.getElementById("mail");
const error = email.nextElementSibling;

// As per the HTML Specification
const emailRegExp =
  /^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$/;
```

```javascript
// Now we can rebuild our validation constraint
// Because we do not rely on CSS pseudo-class, we have to
// explicitly set the valid/invalid class on our email field
window.addEventListener("load", () => {
  // Here, we test if the field is empty (remember, the field is not required)
  // If it is not, we check if its content is a well-formed email address.
  const isValid = email.value.length === 0 || emailRegExp.test(email.value);
  email.className = isValid ? "valid" : "invalid";
});

// This defines what happens when the user types in the field
email.addEventListener("input", () => {
  const isValid = email.value.length === 0 || emailRegExp.test(email.value);
  if (isValid) {
    email.className = "valid";
    error.textContent = "";
    error.className = "error";
  } else {
    email.className = "invalid";
  }
});

// This defines what happens when the user tries to submit the data
form.addEventListener("submit", (event) => {
  event.preventDefault();

  const isValid = email.value.length === 0 || emailRegExp.test(email.value);
  if (!isValid) {
    email.className = "invalid";
    error.textContent = "I expect an email, darling!";
    error.className = "error active";
  } else {
    email.className = "valid";
    error.textContent = "";
    error.className = "error";
  }
});
```

The result looks like this:

</> Play

As you can see, it's not that hard to build a validation system on your own. The difficult part is to make it generic enough to use both cross-platform and on any form you might create. There are many libraries available to perform form validation, such as [Validate.js](#) ↗.

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see [Test your skills: Form validation](#).

Summary

Client-side form validation sometimes requires JavaScript if you want to customize styling and error messages, but it *always* requires you to think carefully about the user. Always remember to help your users correct the data they provide. To that end, be sure to:

- Display explicit error messages.

- Be permissive about the input format.

- Point out exactly where the error occurs, especially on large forms.

Once you have checked that the form is filled out correctly, the form can be submitted. We'll cover [sending form data](#) next.

Advanced Topics

- [How to build custom form controls](#)

- [Sending forms through JavaScript](#)

- [Property compatibility table for form widgets](#)

Help improve MDN

Was this page helpful to you?

👍 Yes   👎 No

[Learn how to contribute](#).

This page was last modified on Jul 25, 2024 by [MDN contributors](#).