Constraint validation

The creation of web forms has always been a complex task. While marking up the form itself is easy, checking whether each field has a valid and coherent value is more difficult, and informing the user about the problem may become a headache. HTML5 introduced new mechanisms for forms: it added new semantic types for the `<input>` element and *constraint validation* to ease the work of checking the form content on the client side. Basic, usual constraints can be checked, without the need for JavaScript, by setting new attributes; more complex constraints can be tested using the Constraint validation API.

For a basic introduction to these concepts, with examples, see the Form validation tutorial.

> **Note:** HTML5 Constraint validation doesn't remove the need for validation on the *server side*. Even though far fewer invalid form requests are to be expected, invalid ones can still be sent by non-compliant browsers (for instance, browsers without HTML5 and without JavaScript) or by bad people trying to trick your web application. Therefore, like with HTML4, you need to also validate input constraints on the server side, in a way that is consistent with what is done on the client side.

Intrinsic and basic constraints

In HTML5, basic constraints are declared in two ways:

- By choosing the most semantically appropriate value for the `type` attribute of the `<input>` element, e.g., choosing the `email` type automatically creates a constraint that checks whether the value is a valid e-mail address.
- By setting values on validation-related attributes, allowing basic constraints to be described in a simple way, without the need for JavaScript.

Semantic input types

The intrinsic constraints for the `type` attribute are:

Input type	Constraint description	Associated violation
`<input type="URL">`	The value must be an absolute URL, as defined in the URL Living Standard .	**TypeMismatch** constraint violation
`<input type="email">`	The value must be a syntactically valid email address, which generally has the format *`username@hostname.tld`*.	**TypeMismatch** constraint violation

For both of these input types, if the `multiple` attribute is set, several values can be set, as a comma-separated list, for this input. If any of these do not satisfy the condition described here, the **Type mismatch** constraint violation is triggered.

Note that most input types don't have intrinsic constraints, as some are barred from constraint validation or have a sanitization algorithm transforming incorrect values to a correct default.

Validation-related attributes

In addition to the `type` attribute described above, the following attributes are used to describe basic constraints:

Attribute	Input types supporting the attribute	Possible values	Constraint description	Associated violation

Attribute	Input types supporting the attribute	Possible values	Constraint description	Associated violation
`pattern`	`text`, `search`, `url`, `tel`, `email`, `password`	A JavaScript regular expression (compiled with the ECMAScript 5 `global`, `ignoreCase`, and `multi-line` flags *disabled)*	The value must match the pattern.	**`pattern-Mismatch`** constraint violation
`min`	`range`, `number`	A valid number	The value must be greater than or equal to the value.	**`rangeUn-derflow`** constraint violation
	`date`, `month`, `week`	A valid date		
	`datetime`, `datetime-local`, `time`	A valid date and time		
`max`	`range`, `number`	A valid number	The value must be less than or equal to the value	**`rangeOver flow`** constraint violation
	`date`, `month`, `week`	A valid date		
	`datetime`, `datetime-local`, `time`	A valid date and time		

Attribute	Input types supporting the attribute	Possible values	Constraint description	Associated violation
required	`text`, `search`, `url`, `tel`, `email`, `password`, `date`, `datetime`, `datetime-local`, `month`, `week`, `time`, `number`, `checkbox`, `radio`, `file`; also on the `<select>` and `<textarea>` elements	*none* as it is a Boolean attribute: its presence means *true*, its absence means *false*	There must be a value (if set).	**`valueMissing`** constraint violation
step	`date`	An integer number of days	Unless the step is set to the `any` literal, the value must be **min** + an integral multiple of the step.	**`stepMismatch`** constraint violation
	`month`	An integer number of months		
	`week`	An integer number of weeks		
	`datetime`, `datetime-local`, `time`	An integer number of seconds		
	`range`, `number`	An integer		

Attribute	Input types supporting the attribute	Possible values	Constraint description	Associated violation
minlength	`text`, `search`, `url`, `tel`, `email`, `password`; also on the `<textarea>` element	An integer length	The number of characters (code points) must not be less than the value of the attribute, if non-empty. All newlines are normalized to a single character (as opposed to CRLF pairs) for `<textarea>`.	`tooShort` constraint violation
maxlength	`text`, `search`, `url`, `tel`, `email`, `password`; also on the `<textarea>` element	An integer length	The number of characters (code points) must not exceed the value of the attribute.	`tooLong` constraint violation

Constraint validation process

Constraint validation is done through the Constraint Validation API either on a single form element or at the form level, on the `<form>` element itself. The constraint validation is done in the following ways:

- By a call to the `checkValidity()` or `reportValidity()` method of a form-associated DOM interface, (`HTMLInputElement`, `HTMLSelectElement`, `HTMLButtonElement`, `HTMLOutputElement` or `HTMLTextAreaElement`), which evaluates the constraints only on this element, allowing a script to get this information. The `checkValidity()` method returns a Boolean indicating whether the element's value passes its constraints. (This is typically done by the user-agent when determining which of the CSS pseudo-classes, `:valid` or `:invalid` applies.) In contrast, the `reportValidity()` method reports any

`.valid` or `.invalid`, applies.) In contrast, the `reportValidity()` method reports any constraint failures to the user.

- By a call to the `checkValidity()` or `reportValidity()` method on the `HTMLFormElement` interface.
- By submitting the form itself.

Calling `checkValidity()` is called *statically* validating the constraints, while calling `reportValidity()` or submitting the form is called *interactively* validating the constraints.

> **Note:**
>
> - If the `novalidate` attribute is set on the `<form>` element, interactive validation of the constraints doesn't happen.
> - Calling the `submit()` method on the `HTMLFormElement` interface doesn't trigger a constraint validation. In other words, this method sends the form data to the server even if doesn't satisfy the constraints. Call the `click()` method on a submit button instead.

Complex constraints using the Constraint Validation API

Using JavaScript and the Constraint API, it is possible to implement more complex constraints, for example, constraints combining several fields, or constraints involving complex calculations.

Basically, the idea is to trigger JavaScript on some form field event (like **onchange**) to calculate whether the constraint is violated, and then to use the method `*field*.setCustomValidity()` to set the result of the validation: an empty string means the constraint is satisfied, and any other string means there is an error and this string is the error message to display to the user.

Constraint combining several fields: Postal code validation

The postal code format varies from one country to another. Not only do most countries allow an optional prefix with the country code (like `D-` in Germany, `F-` in France or Switzerland), but some countries have postal codes with only a fixed number of digits; others, like the UK, have more complex structures, allowing letters at some specific positions.

> **Note:** This is not a comprehensive postal code validation library, but rather a demonstration of

> the key concepts.

As an example, we will add a script checking the constraint validation for this simple form:

```
<form>
    <label for="ZIP">ZIP : </label>
    <input type="text" id="ZIP">
    <label for="Country">Country : </label>
    <select id="Country">
      <option value="ch">Switzerland</option>
      <option value="fr">France</option>
      <option value="de">Germany</option>
      <option value="nl">The Netherlands</option>
    </select>
    <input type="submit" value="Validate">
</form>
```

This displays the following form:

First, we write a function checking the constraint itself:

```
function checkZIP() {
  // For each country, defines the pattern that the ZIP has to follow
  var constraints = {
    ch : [ '^(CH-)?\\d{4}$', "Switzerland ZIPs must have exactly 4 digits: e.g
    fr : [ '^(F-)?\\d{5}$' , "France ZIPs must have exactly 5 digits: e.g. F-7
    de : [ '^(D-)?\\d{5}$' , "Germany ZIPs must have exactly 5 digits: e.g. D-
    nl : [ '^(NL-)?\\d{4}\\s*([A-RT-Z][A-Z]|S[BCE-RT-Z])$',
                    "Nederland ZIPs must have exactly 4 digits, followed by 2
  };

  // Read the country id
  var country = document.getElementById("Country").value;

  // Get the NPA field
  var ZIPField = document.getElementById("ZIP");


  // Build the constraint checker
  var constraint = new RegExp(constraints[country][0], "");
    console.log(constraint);

  // Check it!
  if (constraint.test(ZIPField.value)) {
    // The ZIP follows the constraint, we use the ConstraintAPI to tell it
    ZIPField.setCustomValidity("");
  }
  else {
    // The ZIP doesn't follow the constraint, we use the ConstraintAPI to
    // give a message about the format required for this country
    ZIPField.setCustomValidity(constraints[country][1]);
  }
}
```

Then we link it to the **onchange** event for the `<select>` and the **oninput** event for the `<input>`:

```
window.onload = function () {
    document.getElementById("Country").onchange = checkZIP;
    document.getElementById("ZIP").oninput = checkZIP;
}
```

You can see a live example of the postal code validation.

Limiting the size of a file before its upload

Another common constraint is to limit the size of a file to be uploaded. Checking this on the client side before the file is transmitted to the server requires combining the Constraint Validation API, and especially the `field.setCustomValidity()` method, with another JavaScript API, here the File API.

Here is the HTML part:

```
<label for="FS">Select a file smaller than 75 kB : </label>
```

```
<input type="file" id="FS">
```

This displays:

The JavaScript reads the file selected, uses the `File.size()` method to get its size, compares it to the (hard coded) limit, and calls the Constraint API to inform the browser if there is a violation:

```
function checkFileSize() {
  var FS = document.getElementById("FS");
  var files = FS.files;

  // If there is (at least) one file selected
  if (files.length > 0) {
     if (files[0].size > 75 * 1024) { // Check the constraint
       FS.setCustomValidity("The selected file must not be larger than 75 kB")
       return;
     }
  }
  // No custom constraint violation
  FS.setCustomValidity("");
}
```

Finally we hook the method with the correct event:

```
window.onload = function () {
  document.getElementById("FS").onchange = checkFileSize;
}
```

You can see a [live example](#) of the File size constraint validation.

Visual styling of constraint validation

Apart from setting constraints, web developers want to control what messages are displayed to the users and how they are styled.

Controlling the look of elements

The look of elements can be controlled via CSS pseudo-classes.

:required and :optional CSS pseudo-classes

The `:required` and `:optional` pseudo-classes allow writing selectors that match form elements that have the `required` attribute, or that don't have it.

:placeholder-shown CSS pseudo-class

See `:placeholder-shown`

:valid :invalid CSS pseudo-classes

The `:valid` and `:invalid` pseudo-classes are used to represent <input> elements whose content validates and fails to validate respectively according to the input's type setting. These classes allow the user to style valid or invalid form elements to make it easier to identify elements that are either formatted correctly or incorrectly.

Controlling the text of constraint violation

The following items can help with controlling the text of a constraint violation:

- element.setCustomValidity(message) method on the following elements:
  - `<fieldset>`. Note: Setting a custom validity message on fieldset elements will not prevent form submission in most browsers.
  - `<input>`
  - `<output>`
  - `<select>`
  - Submit buttons (created with either a `<button>` element with the `submit` type, or an `input` element with the submit type. Other types of buttons do not participate in constraint validation.
  - `<textarea>`

- The `ValidityState` interface describes the object returned by the validity property of the element types listed above. It represents various ways that an entered value can be invalid. Together, they help explain why an element's value fails to validate, if it's not valid.

Last modified: Mar 17, 2021, by MDN contributors

Change your language

English (US) Change language