We now look at the functionality of the different form widgets in detail, looking at what options are available to collect different types of data. This guide is somewhat exhaustive, covering all of the available native form widgets.

Prerequisites:	Basic computer literacy, and a basic understanding of HTML.
Objective:	To understand what types of native form widget are available in brow collecting data, and how to implement them using HTML.

Here we will focus on the form widgets built in to browsers, but because HTML forms remain quite limited and the quality of the implementations can be very different from one browser to another, web developers sometimes build their own form widgets — see How to build custom form widgets later in the module for more ideas about this.

Note: Most of the features discussed in this article have wide support across browsers; we'll note exceptions to this. If you want more exact details, you should consult our HTML forms element reference, and in particular our extensive <input> types reference.

Common attributes

Many of the elements used to define form widgets have some of their own attributes. However, there is a set of attributes common to all form elements that give you some control over those widgets. Here is a list of those common attributes:

Attribute name	Default value	Description
`autofocus`	(*false*)	This Boolean attribute lets you specify that the eleme input focus when the page loads, unless the user ove in a different control. Only one form-associated elem attribute specified.
`disabled`	(*false*)	This Boolean attribute indicates that the user cannot attribute is not specified, the element inherits its setti

Attribute name	Default value	Description
		for example, `<fieldset>`; if there is no containing element then the element is enabled.
`form`		The form element that the widget is associated with. Th be the `id` attribute of a `<form>` element in the same docur a form widget outside of a `<form>`element. In practice, ho which supports that feature.
`name`		The name of the element; this is submitted with the form
`value`		The element's initial value.

Text input fields

Text `<input>` fields are the most basic form widgets. They are a very convenient way to let the user enter any kind of data. However, some text fields can be specialized to achieve particular needs. We've already seen a few simple examples

Note: HTML form text fields are simple plain text input controls. This means that you cannot use them to perform rich editing (bold, italic, etc.). All rich text editors you'll encounter out there are custom widgets created with HTML, CSS, and JavaScript.

All text fields share some common behaviors:

- They can be marked as `readonly` (the user cannot modify the input value) or even `disabled` (the input value is never sent with the rest of the form data).
- They can have a `placeholder`; this is text that appears inside the text input box that describes the purpose of the box briefly.
- They can be constrained in `size` (the physical size of the box) and length (the maximum number of characters that can be entered into the box).
- They can benefit from spell checking, if the browser supports it.

Note: The `<input>` element is special because it can be almost anything. By simply setting its `type` attribute, it can change radically, and it is used for creating most types of form widget including single line text fields, controls without text input, time and date controls, and buttons. However, there are some exceptions, like `<textarea>` for multi-line inputs. Take careful note of these as you read the article.

Single line text fields

A single line text field is created using an `<input>` element whose `type` attribute value is set to `text` (also, if you don't provide the `type` attribute, `text` is the default value). The value `text` for this attribute is also the fallback value if the value you specify for the `type`attribute is unknown by the browser (for example if you specify `type="date"` and the browser doesn't support native date pickers).

Note: You can find examples of all the single line text field types on GitHub at single-line-text-fields.html (see it live also).

Here is a basic single line text field example:

```
<input type="text" id="comment" name="comment" value="I'm a text field">
```

Single line text fields have only one true constraint: if you type text with line breaks, the browser removes those line breaks before sending the data.

HTML5 enhances the basic single line text field by adding special values for the `type`attribute. Those values still turn an `<input>` element into a single line text field but they add a few extra constraints and features to the field.

E-mail address field

This type of field is set with the value `email` for the `type` attribute:

```
<input type="email" id="email" name="email" multiple>
```

When this `type` is used, the user is required to type a valid e-mail address into the field; any other content causes the browser to display an error when the form is submitted. Note that this is client-side error validation, performed by the browser:

It's also possible to let the user type several e-mail addresses into the same input (separated by commas) by including the `multiple` attribute.
On some devices (especially on mobile), a different virtual keypad might be presented that is more suitable for entering email addresses.

Note: You can find out more about form validation in the article Form data validation.

Password field

This type of field is set using the value `password` for the `type` attribute:

```
<input type="password" id="pwd" name="pwd">
```

It doesn't add any special constraints to the entered text, but it does obscure the value entered into the field (e.g. with dots or asterisks) so it can't be read by others.

Keep in mind this is just a user interface feature; unless you submit your form securely, it will get sent in plain text, which is bad for security — a malicious party could intercept your data and steal passwords, credit card details, or whatever else you've submitted. The best way to protect users from this is to host any pages involving forms over a secure connection (i.e. at an https:// ... address), so the data is encrypted before it is sent.

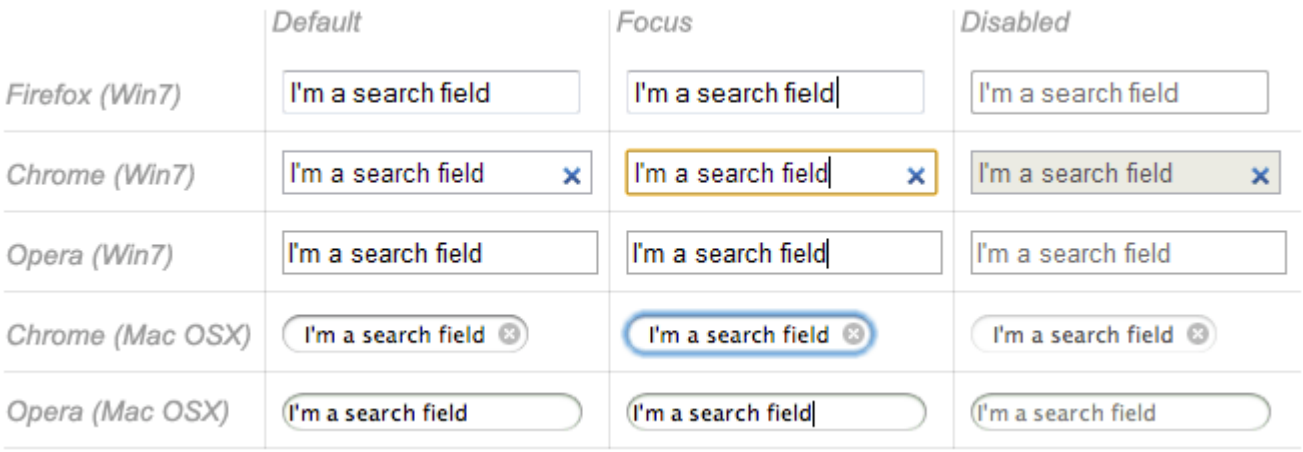
Modern browsers recognize the security implications of sending form data over an insecure connection, and have implemented warnings to deter users from using insecure forms. For more information on what Firefox implements, see Insecure passwords.

Search field

This type of field is set by using the value `search` for the `type` attribute:

```
<input type="search" id="search" name="search">
```

The main difference between a text field and a search field is how the browser styles it — often, search fields are rendered with rounded corners, and/or given an "x" to press to clear the entered value. However, there is another added feature worth noting: their values can be automatically saved to be auto completed across multiple pages on the same site.



Phone number field

This type of field is set using `tel` as the value of the `type` attribute:

```
<input type="tel" id="tel" name="tel">
```

Due to the wide variety of phone number formats around the world, this type of field does not enforce any constraints on the value entered by a user (this can include letters, etc.). This is primarily a semantic difference, although on some devices (especially on mobile), a different virtual keypad might be presented that is more suitable for entering phone numbers.

URL field

This type of field is set using the value `url` for the `type` attribute:

```
<input type="url" id="url" name="url">
```

It adds special validation constraints to the field, with the browser reporting an error if invalid URLs are entered.

Note: Just because the URL is well-formed doesn't necessarily mean that it refers to a location that actually exists.

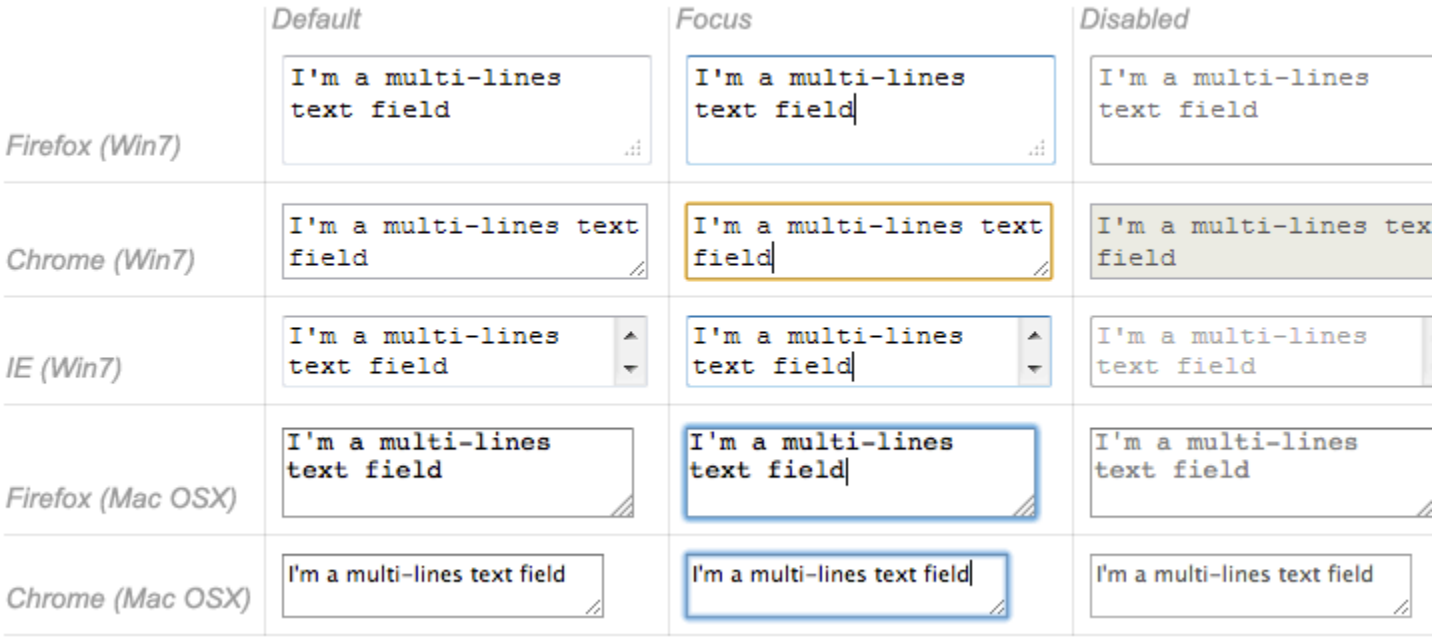
Note: Fields that have special constraints and are in error prevent the form from being sent; in addition, they can be styled so as to make the error clear. We will discuss this in detail in the article: Data form validation.

Multi-line text fields

A multi-line text field is specified using a `<textarea>` element, rather than using the `<input>` element.

```
<textarea cols="30" rows="10"></textarea>
```

The main difference between a textarea and a regular single line text field is that users are allowed to type text that includes hard line breaks (i.e. pressing return).



Note: You can find an example of a multi-line text field on GitHub at multi-line-text-field.html (see it live also). Have a look at it, and notice how in most browsers, the text area is given a drag handle on the bottom right to allow the user to resize it. This resizing ability can be turned off by setting the text area's `resize` property to `none` using CSS.

`<textarea>` also accepts a few extra attributes to control its rendering across several lines (in addition to several others):

Attributes for the `<textarea>` element

Attribute name	Default value	Description
cols	20	The visible width of the text control, in average cha
rows		The number of visible text lines for the control.
wrap	soft	Indicates how the control wraps text. Possible valu

Note that the `<textarea>` element is written a bit differently from the `<input>` element. The `<input>` element is an empty element, which means that it cannot contain any child elements. On the other hand, the `<textarea>` element is a regular element that can contain text content children.
There are two key related points to note here:

- If you want to define a default value for an `<input>` element, you have to use the `value` attribute; for a `<textarea>` element on the other hand you put the default text between the starting tag and the closing tag of the `<textarea>`.
- Because of its nature, the `<textarea>` element only accepts text content; this means that any HTML content put inside a `<textarea>` is rendered as if it was plain text content.

Drop-down content

Drop-down widgets are a simple way to let users select one of many options without taking up much space in the user interface. HTML has two forms of drop down content: the **select box**, and **autocomplete box**. In both cases the interaction is the same — once the control is activated, the browser displays a list of values the user can select between.

Note: You can find examples of all the drop-down box types on GitHub at drop-down-content.html (see it live also).

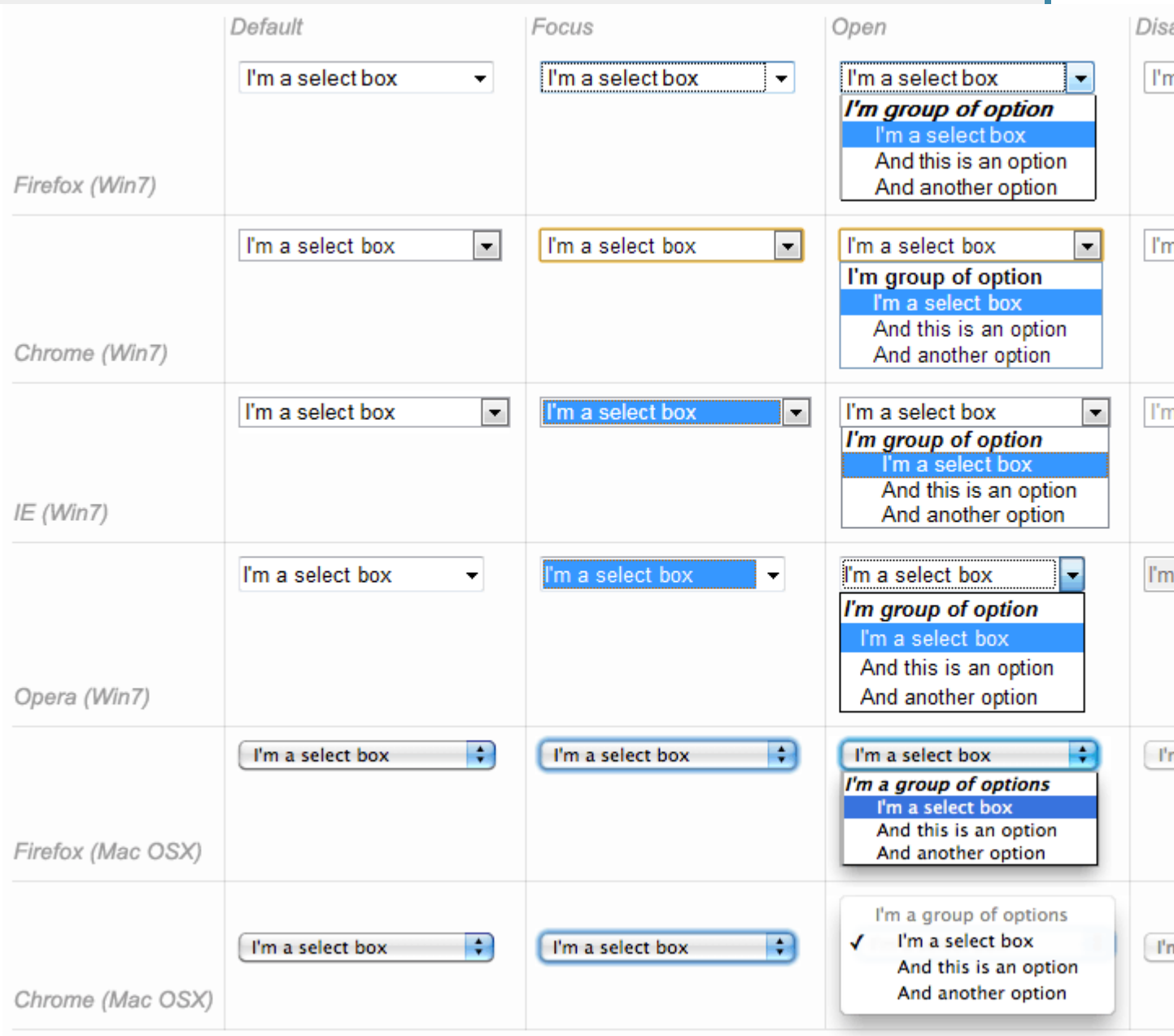
Select box

A select box is created with a `<select>` element with one or more `<option>` elements as its children, each of which specifies one of its possible values.

```
<select id="simple" name="simple">
  <option>Banana</option>
  <option>Cherry</option>
  <option>Lemon</option>
</select>
```

If required, the default value for the select box can be set using the `selected` attribute on the desired `<option>` element — this option is then preselected when the page loads. The `<option>` elements can also be nested inside `<optgroup>` elements to create visually associated groups of values:

```
<select id="groups" name="groups">
  <optgroup label="fruits">
    <option>Banana</option>
    <option selected>Cherry</option>
    <option>Lemon</option>
  </optgroup>
  <optgroup label="vegetables">
    <option>Carrot</option>
    <option>Eggplant</option>
    <option>Potato</option>
  </optgroup>
</select>
```



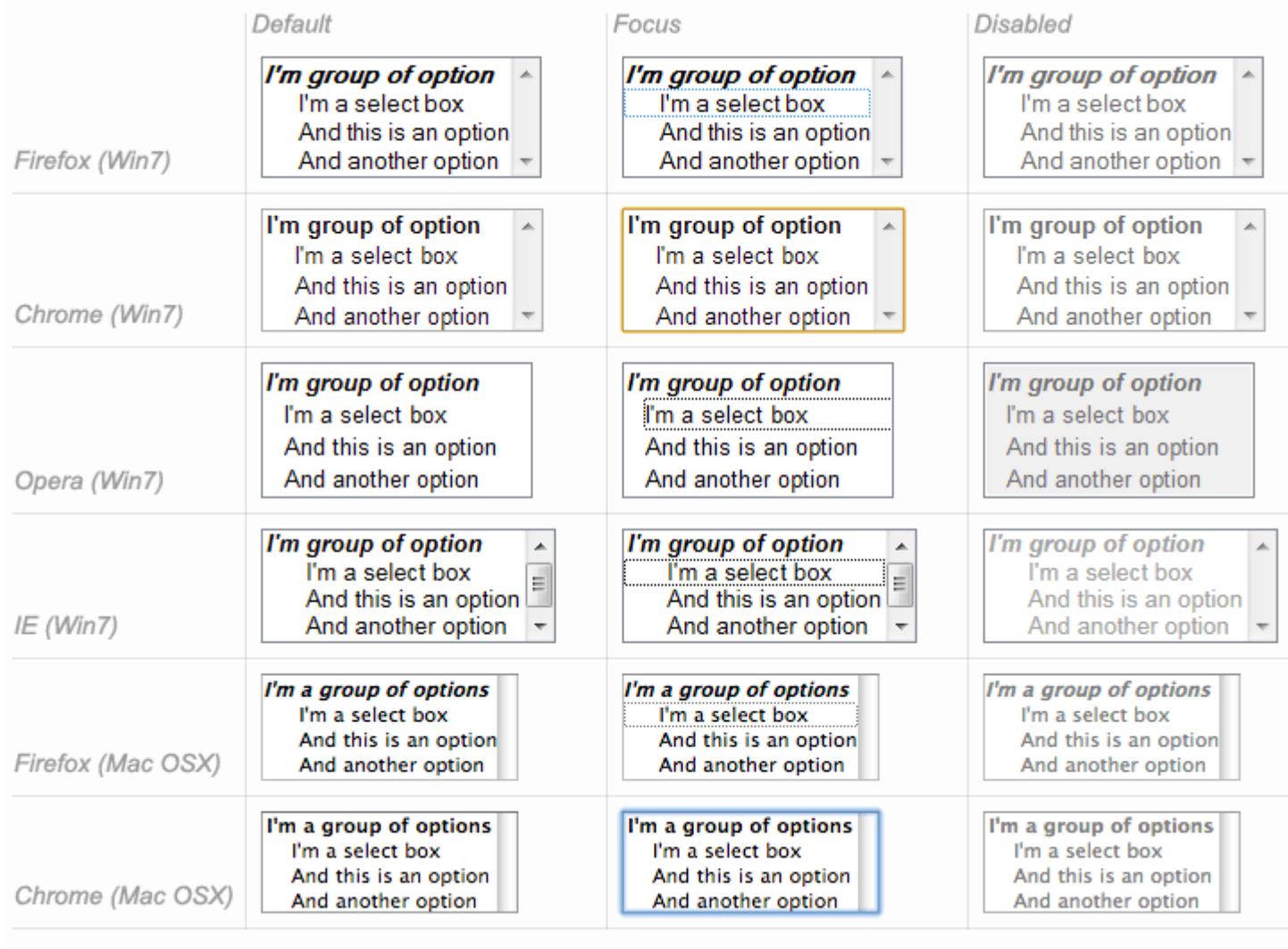
If an `<option>` element is set with a `value` attribute, that attribute's value is sent when the form is submitted. If the `value` attribute is omitted, the content of the `<option>`element is used as the select box's value.
On the `<optgroup>` element, the `label` attribute is displayed before the values, but even if it looks somewhat like an option, it is not selectable.

Multiple choice select box

By default, a select box only lets the user select a single value. By adding the `multiple`attribute to the `<select>` element, you can allow users to select several values, by using the default mechanism provided by the operating system (e.g. holding down **`Cmd`**/**`Ctrl`** and clicking multiple values).

Note: In the case of multiple choice select boxes, the select box no longer displays the values as drop-down content — instead, they are all displayed at once in a list.

```
<select multiple id="multi" name="multi">
  <option>Banana</option>
  <option>Cherry</option>
  <option>Lemon</option>
</select>
```



Note: All browsers that support the `<select>` element also support the `multiple`attribute on it.

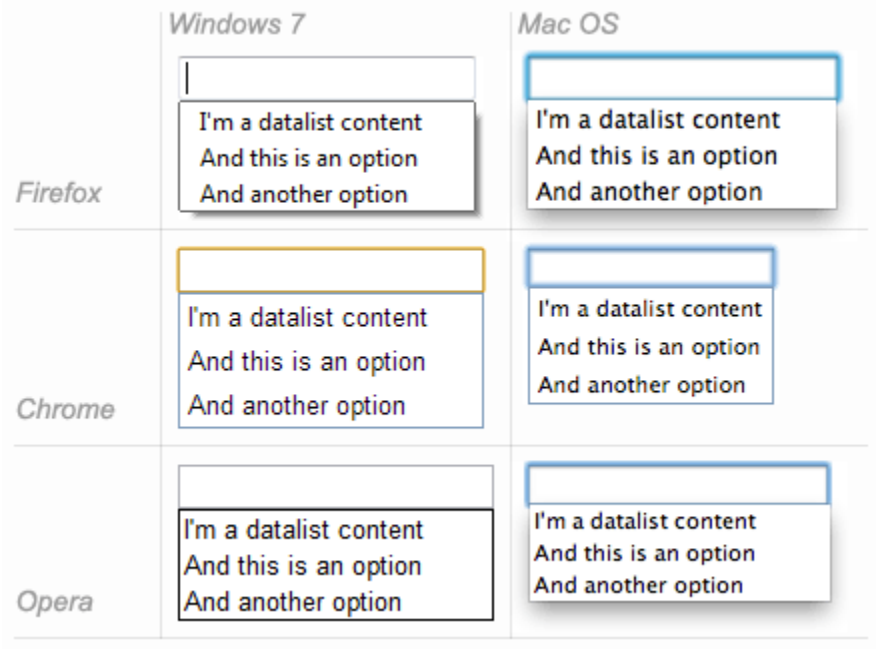
Autocomplete box

You can provide suggested, automatically-completed values for form widgets using the `<datalist>` element with some child `<option>` elements to specify the values to display.
The data list is then bound to a text field (usually an `<input>` element) using the `list`attribute.
Once a data list is affiliated with a form widget, its options are used to auto-complete text entered by the user; typically, this is presented to the user as a drop-down box listing possible matches for what they've typed into the input.

```
<label for="myFruit">What's your favorite fruit?</label>
```

```
<input type="text" name="myFruit" id="myFruit" list="mySuggestion">
<datalist id="mySuggestion">
  <option>Apple</option>
  <option>Banana</option>
  <option>Blackberry</option>
  <option>Blueberry</option>
  <option>Lemon</option>
  <option>Lychee</option>
  <option>Peach</option>
  <option>Pear</option>
</datalist>
```

Note: According to the HTML specification, the `list` attribute and the `<datalist>` element can be used with any kind of widget requiring a user input. However, it is unclear how it should work with controls other than text (color or date for example), and different browsers behave differently from case to case. Because of that, be cautious using this feature with anything but text fields.



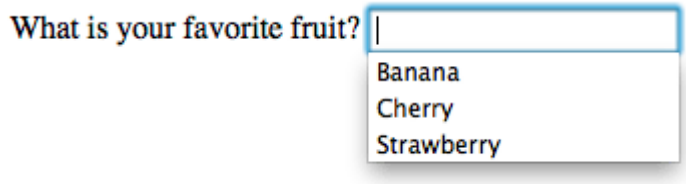
Datalist support and fallbacks

The `<datalist>` element is a very recent addition to HTML forms, so browser support is a bit more limited than what we saw earlier. Most notably, it isn't supported in IE versions below 10, and Safari still doesn't support it at the time of writing.

To handle this, here is a little trick to provide a nice fallback for those browsers:

```
<label for="myFruit">What is your favorite fruit? (With fallback)</label>
<input type="text" id="myFruit" name="fruit" list="fruitList">

<datalist id="fruitList">
  <label for="suggestion">or pick a fruit</label>
  <select id="suggestion" name="altFruit">
    <option>Apple</option>
    <option>Banana</option>
    <option>Blackberry</option>
    <option>Blueberry</option>
    <option>Lemon</option>
    <option>Lychee</option>
    <option>Peach</option>
    <option>Pear</option>
  </select>
</datalist>
```

Browsers that support the `<datalist>` element will ignore all the elements that are not `<option>` elements and will work as expected. On the other hand, browsers that do not support the `<datalist>` element will display the label and the select box. Of course, there are other ways to handle the lack of support for the `<datalist>` element, but this is the simplest (others tend to require JavaScript).

Safari 6	
Firefox 18	

Checkable items

Checkable items are widgets whose state you can change by clicking on them. There are two kinds of checkable item: the check box and the radio

button. Both use the `checked`attribute to indicate whether the widget is checked by default or not.
It's worth noting that these widgets do not behave exactly like other form widgets. For most form widgets, once the form is submitted all widgets that have a `name` attribute are sent, even if no value has been filled out. In the case of checkable items, their values are sent only if they are checked. If they are not checked, nothing is sent, not even their name.

Note: You can find the examples from this section on GitHub as checkable-items.html (see it live also).

For maximum usability/accessibility, you are advised to surround each list of related items in a `<fieldset>`, with a `<legend>` providing an overall description of the list. Each individual pair of `<label>`/`<input>` elements should be contained in its own list item (or similar). This is shown in the examples.
You also need to provide values for these kinds of inputs inside the `value` attribute if you want them to be meaningful — if no value is provided, check boxes and radio buttons are given a value of `on`.

Check box

A check box is created using the `<input>` element with its `type` attribute set to the value `checkbox`.

```
<input type="checkbox" checked id="carrots" name="carrots" value="carrots">
```

Including the `checked` attribute makes the checkbox checked automatically when the page loads.

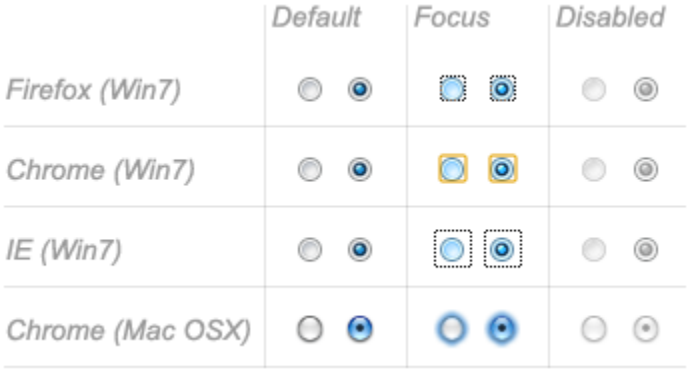
Radio button

A radio button is created using the `<input>` element with its `type` attribute set to the value `radio`.

```
<input type="radio" checked id="soup" name="meal">
```

Several radio buttons can be tied together. If they share the same value for their `name`attribute, they will be considered to be in the same group of buttons. Only one button in a given group may be checked at the same time; this means that when one of them is checked all the others automatically get unchecked. When the form is sent, only the value of the checked radio button is sent. If none of them are checked, the whole pool of radio buttons is considered to be in an unknown state and no value is sent with the form.

```
<fieldset>
  <legend>What is your favorite meal?</legend>
  <ul>
    <li>
      <label for="soup">Soup</label>
      <input type="radio" checked id="soup" name="meal" value="soup">
    </li>
    <li>
      <label for="curry">Curry</label>
      <input type="radio" id="curry" name="meal" value="curry">
    </li>
    <li>
      <label for="pizza">Pizza</label>
      <input type="radio" id="pizza" name="meal" value="pizza">
    </li>
  </ul>
</fieldset>
```



Buttons

Within HTML forms, there are three kinds of button:

Submit

Sends the form data to the server.

Reset

Resets all form widgets to their default values.

Anonymous

Buttons that have no automatic effect but can be customized using JavaScript code. If you omit the `type` attribute, this is the default value.

Note: You can find the examples from this section on GitHub as button-examples.html (see it live also).

A button is created using a `<button>` element or an `<input>` element. It's the value of the `type` attribute that specifies what kind of button is displayed:

submit

```
<button type="submit">
    This a <br><strong>submit button</strong>
</button>

<input type="submit" value="This is a submit button">
```

reset

```
<button type="reset">
    This a <br><strong>reset button</strong>
</button>

<input type="reset" value="This is a reset button">
```

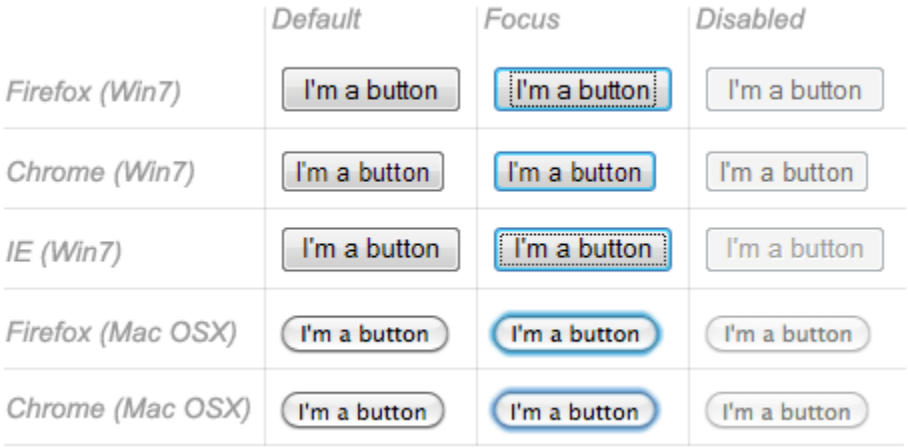
anonymous

```
<button type="button">
    This an <br><strong>anonymous button</strong>
</button>

<input type="button" value="This is an anonymous button">
```

Buttons always behave the same whether you use a `<button>` element or an `<input>`element. There are, however, some notable differences:

- As you can see from the examples, `<button>` elements let you use HTML content in their labels, which are inserted inside the opening and closing `<button>` tags. `<input>` elements on the other hand are empty elements; their labels are inserted inside `value` attributes, and therefore only accept plain text content.

- With `<button>` elements, it's possible to have a value different than the button's label (by setting it inside a `value` attribute). This isn't reliable in versions of Internet Explorer prior to IE 8.



Technically speaking, there is almost no difference between a button defined with the `<button>` element or the `<input>` element. The only noticeable difference is the label of the button itself. Within an `<input>` element, the label can only be character data, whereas in a `<button>` element, the label can be HTML, so it can be styled accordingly.

Advanced form widgets

In this section we cover those widgets that let users input complex or unusual data. This includes exact or approximate numbers, dates and times, or colors.

Note: You can find the examples from this section on GitHub as advanced-examples.html (see it live also).

Numbers

Widgets for numbers are created with the `<input>` element, with its `type` attribute set to the value `number`. This control looks like a text field but allows only floating-point numbers, and usually provides some buttons to increase or decrease the value of the widget.
It's also possible to:

- Constrain the value by setting the `min` and `max` attributes.
- Specify the amount by which the increase and decrease buttons change the widget's value by setting the `step` attribute.

Example

```
<input type="number" name="age" id="age" min="1" max="10" step="2">
```

This creates a number widget whose value is restricted to any value between 1 and 10, and whose increase and decrease buttons change its value by 2.

`number` inputs are not supported in versions of Internet Explorer below 10.

Sliders

Another way to pick a number is to use a slider. Visually speaking, sliders are less accurate than text fields, therefore they are used to pick a number whose exact value is not necessarily important.

A slider is created by using the `<input>` with its `type` attribute set to the value `range`. It's important to properly configure your slider; to that end, it's highly recommended that you set the `min`, `max`, and `step` attributes.

Example

```
<input type="range" name="beans" id="beans" min="0" max="500" step="10">
```

This example creates a slider whose value may range between 0 and 500, and whose increment/decrement buttons change the value by +10 and -10.

One problem with sliders is that they don't offer any kind of visual feedback as to what the current value is. You need to add this yourself with JavaScript, but this is relatively easy to do. In this example we add an empty `<span>` element, in which we will write the current value of the slider, updating it as it is changed.

```
<label for="beans">How many beans can you eat?</label>
<input type="range" name="beans" id="beans" min="0" max="500" step="10">
<span class="beancount"></span>
```

This can be implemented using some simple JavaScript:

```
var beans = document.querySelector('#beans');
var count = document.querySelector('.beancount');

count.textContent = beans.value;

beans.oninput = function() {
  count.textContent = beans.value;
}
```

Here we store references to the range input and the span in two variables, then we immediately set the span's `textContent` to the current `value` of the input. Finally, we set up an `oninput` event handler so that every time the range slider is moved, the span `textContent` is updated to the new input value.
`range` inputs are not supported in versions of Internet Explorer below 10.

Date and time picker

Gathering date and time values has traditionally been a nightmare for web developers. HTML5 brings some enhancements here by providing a special control to handle this specific kind of data.

A date and time control is created using the `<input>` element and an appropriate value for the `type` attribute, depending on whether you wish to collect dates, times, or both.

`datetime-local`

This creates a widget to display and pick a date with time, but without any specific time zone information.

```
<input type="datetime-local" name="datetime" id="datetime">
```

`month`

This creates a widget to display and pick a month with a year.

```
<input type="month" name="month" id="month">
```

`time`

This creates a widget to display and pick a time value.

```
<input type="time" name="time" id="time">
```

`week`

This creates a widget to display and pick a week number and its year.

```
<input type="week" name="week" id="week">
```

All date and time control can be constrained using the `min` and `max` attributes.

```
<label for="myDate">When are you available this summer?</label>
<input type="date" name="myDate" min="2013-06-01" max="2013-08-31" id="myDate">
```

Warning — The date and time widgets are still poorly supported. At the moment, Chrome, Edge and Opera support them well, but there is no support in Internet Explorer, and Firefox and Safari have very little support for these.

Color picker

Colors are always a bit difficult to handle. There are many ways to express them: RGB values (decimal or hexadecimal), HSL values, keywords, etc. The color widget lets users pick a color in both textual and visual ways.

A color widget is created using the `<input>` element with its `type` attribute set to the value `color`.

```
<input type="color" name="color" id="color">
```

Warning — Color widget support it currently not very good. There is no support in Internet Explorer, and Safari currently doesn't support it either. The other major browsers do support it.

Other widgets

There are a few other widgets that cannot be easily classified due to their very specific behaviors, but which are still very useful.

Note: You can find the examples from this section on GitHub as other-examples.html (see it live also).

File picker

HTML forms are able to send files to a server; this specific action is detailed in the article Sending and retrieving form data. The file picker widget is how the user can choose one or more files to send.

To create a file picker widget, you use the `<input>` element with its `type` attribute set to `file`. The types of files that are accepted can be constrained using the `accept` attribute. In addition, if you want to let the user pick more than one file, you can do so by adding the `multiple` attribute.

Example

In this example, a file picker is created that requests graphic image files. The user is allowed to select multiple files in this case.

```
<input type="file" name="file" id="file" accept="image/*" multiple>
```

Hidden content

It's sometimes convenient for technical reasons to have pieces of data that are sent with a form but not displayed to the user. To do this, you can add an invisible element in your form. Use an `<input>` with its `type` attribute set to the value `hidden`.
If you create such an element, it's required to set its `name` and `value` attributes:

```
<input type="hidden" id="timestamp" name="timestamp" value="1286705410">
```

Image button

The **image button** control is one which is displayed exactly like an `<img>` element, except that when the user clicks on it, it behaves like a submit button (see above).
An image button is created using an `<input>` element with its `type` attribute set to the value `image`. This element supports exactly the same set of attributes as the `<img>`element, plus all the attributes supported by other form buttons.

```
<input type="image" alt="Click me!" src="my-img.png" width="80" height="30" />
```

If the image button is used to submit the form, this widget doesn't submit its value; instead the X and Y coordinates of the click on the image are submitted (the coordinates are relative to the image, meaning that the upper-left corner of the image represents the coordinate 0, 0). The coordinates are sent as two key/value pairs:

- The X value key is the value of the `name` attribute followed by the string "*.x*".
- The Y value key is the value of the `name` attribute followed by the string "*.y*".

So for example when you click on the image of this widget, you are sent to a URL like the following:

```
http://foo.com?pos.x=123&pos.y=456
```

This is a very convenient way to build a "hot map". How these values are sent and retrieved is detailed in the Sending and retrieving form data article.

Meters and progress bars

Meters and progress bars are visual representations of numeric values.

Progress

A progress bar represents a value that changes over time up to a maximum value specified by the `max` attribute. Such a bar is created using a `<progress>` element.

```
<progress max="100" value="75">75/100</progress>
```

This is for implementing anything requiring progress reporting, such as the percentage of total files downloaded, or the number of questions filled in on a questionnaire.

The content inside the `<progress>` element is a fallback for browsers that don't support the element and for assistive technologies to vocalize it.

Meter

A meter bar represents a fixed value in a range delimited by a `min` and a `max` value. This value is visualy rendered as a bar, and to know how this bar looks, we compare the value to some other set values:

- The `low` and `high` values divide the range in three parts:
  - The lower part of the range is between the `min` and `low` values (including those values).
  - The medium part of the range is between the `low` and `high` values (excluding those values).
  - The higher part of the range is between the `high` and `max` values (including those values).
- The `optimum` value defines the optimum value for the `<meter>` element. In conjuction with the `low` and `high` value, it defines which part of the range is prefered:
  - If the `optimum` value is in the lower part of the range, the lower range is considered to be the prefered part, the medium range is considered to be the average part and the higher range is considered to be the worst part.

- If the `optimum` value is in the medium part of the range, the lower range is considered to be an average part, the medium range is considered to be the prefered part and the higher range is considered to be average as well.
- If the `optimum` value is in the higher part of the range, the lower range is considered to be the worst part, the medium range is considered to be the average part and the higher range is considered to be the prefered part.

All browsers that implement the `<meter>` element use those values to change the color of the meter bar:

- If the current value is in the prefered part of the range, the bar is green.
- If the current value is in the average part of the range, the bar is yellow.
- If the current value is in the worst part of the range, the bar is red.

Such a bar is created using a `<meter>` element. This is for implementing any kind of meter, for example a bar showing total space used on a disk, which turns red when it starts to get full.

```
<meter min="0" max="100" value="75" low="33" high="66" optimum="50">75</meter>
```

The content inside the `<meter>` element is a fallback for browsers that don't support the element and for assistive technologies to vocalize it.
Support for progress and meter is fairly good — there is no support in Internet Explorer, but other browsers support it well.

Conclusion

As you'll have seen above, there are a lot of different types of available form elements — you don't need to remember all of these details at once, and can return to this article as often as you like to check up on details.

See also

To dig into the different form widgets, there are some useful external resources you should check out:

- The Current State of HTML5 Forms by Wufoo
- HTML5 Tests - inputs on Quirksmode (also available for mobile browsers)

Previous Overview: FormsNext

This article looks at what happens when a user submits a form — where does the data go, and how do we handle it when it gets there? We also look at some of the security concerns associated with sending form data.

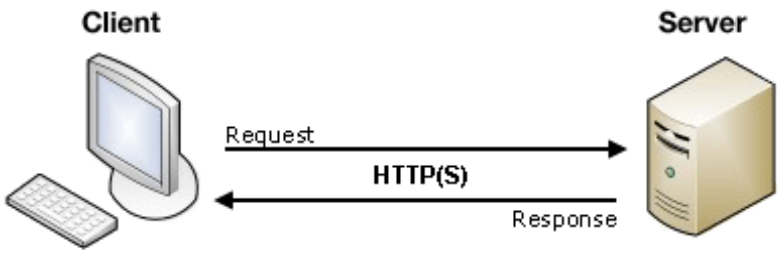
Prerequisites:	Basic computer literacy, an understanding of HTML, and basic knowledge of HTTP and server-side programming.
Objective:	To understand what happens when form data is submitted, including ge idea of how data is processed on the server

Where does the data go?

Here we'll discuss what happens to the data when a form is submitted.

About client/server architecture

The web is based on a very basic client/server architecture that can be summarized as follows: a client (usually a Web browser) sends a request to a server (most of the time a web server like Apache, Nginx, IIS, Tomcat, etc.), using the HTTP protocol. The server answers the request using the same protocol.



On the client side, an HTML form is nothing more than a convenient user-friendly way to configure an HTTP request to send data to a server. This enables the user to provide information to be delivered in the HTTP request.

Note: To get a better idea of how client-server architectures work, read our Server-side website programming first steps module.

On the client side: defining how to send the data

The `<form>` element defines how the data will be sent. All of its attributes are designed to let you configure the request to be sent when a user hits a submit button. The two most important attributes are `action` and `method`.

The `action` attribute

This attribute defines where the data gets sent. Its value must be a valid URL. If this attribute isn't provided, the data will be sent to the URL of the page containing the form.

In this example, the data is sent to an absolute URL — `http://foo.com`:

```
<form action="http://foo.com">
```

Here, we use a relative URL — the data is sent to a different URL on the server:

```
<form action="/somewhere_else">
```

When specified with no attributes, as below, the `<form>` data is sent to the same page that the form is present on:

```
<form>
```

Many older pages use the following notation to indicate that the data should be sent to the same page that contains the form; this was required because until HTML5, the `action`attribute was required. This is no longer needed.

```
<form action="#">
```

Note: It's possible to specify a URL that uses the HTTPS (secure HTTP) protocol. When you do this, the data is encrypted along with the rest of the request, even if the form itself is hosted on an insecure page accessed using HTTP. On the other hand, if the form is hosted on a secure page but you specify an insecure HTTP URL with the `action` attribute, all browsers display a security warning to the user each time they try to send data because the data will not be encrypted.

The `method` attribute

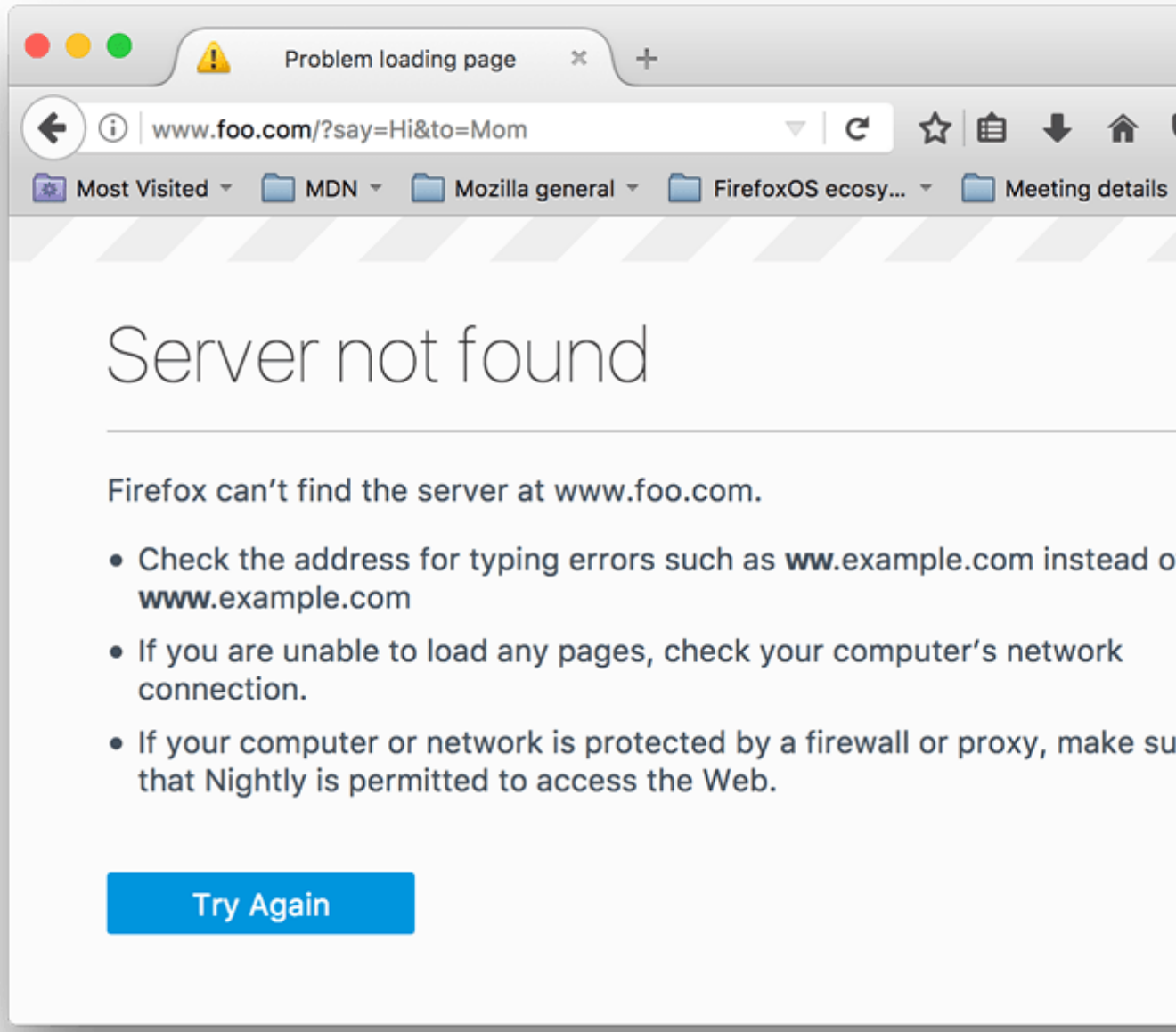
This attribute defines how data is sent. The HTTP protocol provides several ways to perform a request; HTML form data can be transmitted via a number of different ones, the most common of which are the `GET` method and the `POST` method.
To understand the difference between those two methods, let's step back and examine how HTTP works. Each time you want to reach a resource on the Web, the browser sends a request to a URL. An HTTP request consists of two parts: a header that contains a set of global metadata about the browser's capabilities, and a body that can contain information necessary for the server to process the specific request.

The GET method

The `GET` method is the method used by the browser to ask the server to send back a given resource: "Hey server, I want to get this resource." In this case, the browser sends an empty body. Because the body is empty, if a form is sent using this method the data sent to the server is appended to the URL. Consider the following form:

```
<form action="http://foo.com" method="get">
  <div>
    <label for="say">What greeting do you want to say?</label>
    <input name="say" id="say" value="Hi">
  </div>
  <div>
    <label for="to">Who do you want to say it to?</label>
    <input name="to" id="to" value="Mom">
  </div>
  <div>
    <button>Send my greetings</button>
  </div>
</form>
```

Since the `GET` method has been used, you'll see the URL `www.foo.com/?say=Hi&to=Mom`appear in the browser address bar when you submit the form.



The data is appended to the URL as a series of name/value pairs. After the URL web address has ended, we include a question mark (`?`) followed by the name/value pairs, each one separated by an ampersand (`&`). In this case we are passing two pieces of data to the server:

- `say`, which has a value of `Hi`
- `to`, which has a value of `Mom`

The HTTP request looks like this:

```
GET /?say=Hi&to=Mom HTTP/1.1
Host: foo.com
```

Note: You can find this example on GitHub — see get-method.html (see it live also).

The POST method

The `POST` method is a little different. It's the method the browser uses to talk to the server when asking for a response that takes into account the data provided in the body of the HTTP request: "Hey server, take a look at this data and send me back an appropriate result." If a form is sent using this method, the data is appended to the body of the HTTP request.

Let's look at an example — this is the same form we looked at in the `GET` section above, but with the `method` attribute set to `post`.

```
<form action="http://foo.com" method="post">
  <div>
    <label for="say">What greeting do you want to say?</label>
    <input name="say" id="say" value="Hi">
  </div>
  <div>
    <label for="to">Who do you want to say it to?</label>
    <input name="to" id="to" value="Mom">
  </div>
  <div>
    <button>Send my greetings</button>
  </div>
</form>
```

When the form is submitted using the `POST` method, you get no data appended to the URL, and the HTTP request looks like so, with the data included in the request body instead:

```
POST / HTTP/1.1
Host: foo.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 13

say=Hi&to=Mom
```

The `Content-Length` header indicates the size of the body, and the `Content-Type` header indicates the type of resource sent to the server. We'll discuss these headers later on.

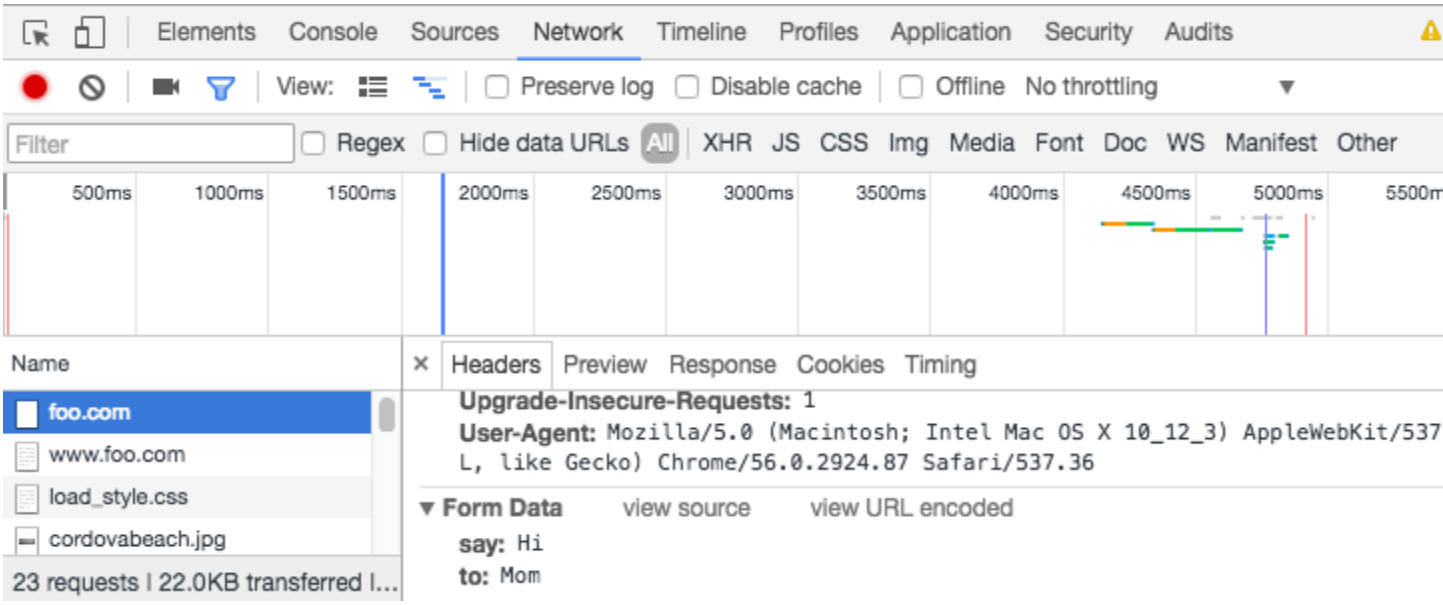
Note: You can find this example on GitHub — see post-method.html (see it live also).

Viewing HTTP requests

HTTP requests are never displayed to the user (if you want to see them, you need to use tools such as the Firefox Network Monitor or the Chrome Developer Tools). As an example, your form data will be shown as follows in the Chrome Network tab. After submitting the form:

1. Press F12
2. Select "Network"
3. Select "All"
4. Select "foo.com" in the "Name" tab
5. Select "Headers"

You can then get the form data, as shown in the image below.



The only thing displayed to the user is the URL called. As we mentioned above, with a `GET`request the user will see the data in their URL bar, but with a `POST` request they won't. This can be very important for two reasons:

1. If you need to send a password (or any other sensitive piece of data), never use the `GET` method or you risk displaying it in the URL bar, which would be very insecure.
2. If you need to send a large amount of data, the `POST` method is preferred because some browsers limit the sizes of URLs. In addition, many servers limit the length of URLs they accept.

On the server side: retrieving the data

Whichever HTTP method you choose, the server receives a string that will be parsed in order to get the data as a list of key/value pairs. The way you access this list depends on the development platform you use and on any

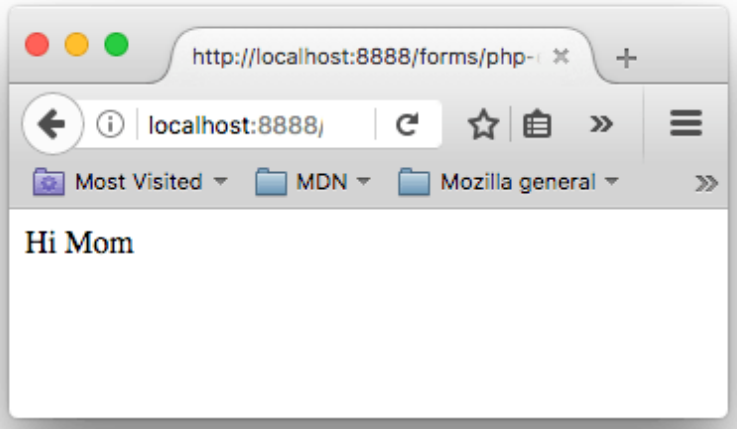
specific frameworks you may be using with it. The technology you use also determines how duplicate keys are handled; often, the most recently received value for a given key is given priority.

Example: Raw PHP

PHP offers some global objects to access the data. Assuming you've used the `POST`method, the following example just takes the data and displays it to the user. Of course, what you do with the data is up to you. You might display it, store it into a database, send it by email, or process it in some other way.

```
<?php
  // The global $_POST variable allows you to access the data sent with the POST
method by name
  // To access the data sent with the GET method, you can use $_GET
  $say = htmlspecialchars($_POST['say']);
  $to  = htmlspecialchars($_POST['to']);

  echo  $say, ' ', $to;
?>
```

This example displays a page with the data we sent. You can see this in action in our example php-example.html file — which contains the same example form as we saw before, with a `method` of `post` and an `action` of `php-example.php`. When it is submitted, it sends the form data to php-example.php, which contains the PHP code seen in the above block. When this code is executed, the output in the browser is `Hi Mom`.



Note: This example won't work when you load it into a browser locally — browsers cannnot interpret PHP code, so when the form is submitted the browser will just offer to download the PHP file for you. To get it to work, you need to run the example through a PHP server of some kind. Good options for local PHP testing are MAMP (Mac and Windows) and AMPPS (Mac, Windows, Linux).

Example: Python

This example shows how you would use Python to do the same thing — display the submitted data on a web page. This uses the Flask framework for rendering the templates, handling the form data submission, etc (see python-example.py).

```
from flask import Flask, render_template, request
app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def form():
    return render_template('form.html')

@app.route('/hello', methods=['GET', 'POST'])
def hello():
    return render_template('greeting.html', say=request.form['say'],
to=request.form['to'])

if __name__ == "__main__":
    app.run()
```

The two templates referenced in the above code are as follows:

- form.html: The same form as we saw above in the The POST method section but with the `action` set to `{{ url_for('hello') }}`. (This is a Jinja2 template, which is basically HTML but can contain calls to the Python code that is running the web server contained in curly braces. `url_for('hello')` is basically saying "redirect to `/hello`when the form is submitted".)
- greeting.html: This template just contains a line that renders the two bits of data passed to it when it is rendered. This is done via the `hello()` function seen above, which runs when the `/hello` URL is navigated to.

Note: Again, this code won't work if you just try to load it into a browser directly. Python works a bit differently to PHP — to run this code locally you'll need to install Python/PIP, then install Flask using `pip3 install flask`. At this point you should be able to run the example using `python3 python-example.py`, then navigating to `localhost:5000` in your browser.

Other languages and frameworks

There are many other server-side technologies you can use for form handling, including Perl, Java, .Net, Ruby, etc. Just pick the one you like best. That said, it's worth noting that it's very uncommon to use these technologies directly because this can be tricky. It's more common to use one of the many nice frameworks that make handling forms easier, such as:

- Symfony for PHP
- Django for Python (a bit more heavyweight than Flask, but with more tools and options).
- Express for Node.js
- Ruby On Rails for Ruby
- Grails for Java
- etc.

It's worth noting that even using these frameworks, working with forms isn't necessarily *easy*. But it's much easier than trying to write all the functionality yourself from scratch, and will save you a lot of time.

Note: It is beyond the scope of this article to teach you any server-side languages or frameworks. The links above will give you some help, should you wish to learn them.

A special case: sending files

Sending files with HTML forms is a special case. Files are binary data — or considered as such — whereas all other data is text data. Because HTTP is a text protocol, there are special requirements for handling binary data.

The `enctype` attribute

This attribute lets you specify the value of the `Content-Type` HTTP header included in the request generated when the form is submitted. This header is very important because it tells the server what kind of data is being sent. By default, its value is `application/x-www-form-urlencoded`. In human terms, this means: "This is form data that has been encoded into URL parameters."
If you want to send files, you need to take three extra steps:

- Set the `method` attribute to `POST` because file content can't be put inside URL parameters.
- Set the value of `enctype` to `multipart/form-data` because the data will be split into multiple parts, one for each file plus one for the text data included in the form body (if text is also entered into the form).
- Include one or more File picker widgets to allow your users to select the file(s) that will be uploaded.
  For example:

```
<form method="post" enctype="multipart/form-data">
  <div>
    <label for="file">Choose a file</label>
    <input type="file" id="file" name="myFile">
  </div>
  <div>
    <button>Send the file</button>
  </div>
</form>
```

Note: Some browsers support the `multiple` attribute on the `<input>` element, which allows more than one file to be chosen for uploading with only one `<input>`element. How the server handles those files really depends on the technology used on the server. As mentioned previously, using a framework will make your life a lot easier.

Warning: Many servers are configured with a size limit for files and HTTP requests in order to prevent abuse. It's important to check this limit with the server administrator before sending a file.

Common security concerns

Each time you send data to a server, you need to consider security. HTML forms are by far the most common attack vectors (places where attacks can occur) against servers. The problems never come from the HTML forms themselves — they come from how the server handles data.

Depending on what you're doing, there are some very well-known security issues that you'll come up against:

XSS and CSRF

Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) are common types of attacks that occur when you display data sent by a user back to the user or to another user.

XSS lets attackers inject client-side script into Web pages viewed by other users. A cross-site scripting vulnerability may be used by attackers to bypass access controls such as the same origin policy. The effect of these attacks may range from a petty nuisance to a significant security risk.
CSRF attacks are similar to XSS attacks in that they start the same way — by injecting client-side script into Web pages — but their target is different. CSRF attackers try to escalate privileges to those of a higher-privileged user (such as a site administrator) to perform an action they shouldn't be able to do (for example, sending data to an untrusted user).

XSS attacks exploit the trust a user has for a web site, while CSRF attacks exploit the trust a web site has for its users.

To prevent these attacks, you should always check the data a user sends to your server and (if you need to display it) try not to display HTML content as provided by the user. Instead, you should process the user-provided data so you don't display it verbatim. Almost all frameworks on the market today implement a minimal filter that removes the
HTML `<script>`, `<iframe>` and `<object>` elements from data sent by any user. This helps to mitigate the risk, but doesn't necessarily eradicate it.

SQL injection

SQL injection is a type of attack that tries to perform actions on a database used by the target web site. This typically involves sending a SQL request in the hope that the server will execute it (usually when the application server tries to store data sent by a user). This is actually one of the main vector attacks against web sites.
The consequences can be terrible, ranging from data loss to attacks taking control of a whole website infrastructure by using privilege escalation. This is a very serious threat and you should never store data sent by a user without performing some sanitization (for example, by
using `mysql_real_escape_string()` on a PHP/MySQL infrastructure).

HTTP header injection and email injection

These kinds of attacks can occur when your application builds HTTP headers or emails based on the data input by a user on a form. These won't directly damage your server or affect your users, but they are an open door to deeper problems such as session hijacking or phishing attacks.

These attacks are mostly silent, and can turn your server into a zombie.

Be paranoid: Never trust your users

So, how do you fight these threats? This is a topic far beyond this guide, but there are a few rules to keep in mind. The most important rule is: never ever trust your users, including yourself; even a trusted user could have been hijacked.

All data that comes to your server must be checked and sanitized. Always. No exception.

- Escape potentially dangerous characters. The specific characters you should be cautious with vary depending on the context in which the data is used and the server platform you employ, but all server-side languages have functions for this.
- Limit the incoming amount of data to allow only what's necessary.
- Sandbox uploaded files (store them on a different server and allow access to the file only through a different subdomain or even better through a fully different domain name).

  You should avoid many/most problems if you follow these three rules, but it's always a good idea to get a security review performed by a competent third party. Don't assume that you've seen all the possible problems.

Note: The Website security article of our server-side learning topic discusses the above threats and potential solutions in more detail.

Conclusion

As you can see, sending form data is easy, but securing an application can be tricky. Just remember that a front-end developer is not the one who should define the security model of the data. Yes, as we'll see, it's possible to perform client side data validation but the server can't trust this validation because it has no way to truly know what really happens on the client side.

Sending data is not enough — we also need to make sure that the data users fill out in forms is in the correct format that we need to process it successfully, and that it won't break our applications. We also want to help our users to fill out our forms correctly and not get frustrated when trying to use our apps. Form validation helps us achieve these goals — this article tells you what you need to know.

Prerequisites:	Computer literacy, a reasonable understanding of HTML, CSS, and Java
Objective:	To understand what form validation is, why it's important, and how to im

What is form validation?

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

- "This field is required" (you can't leave this field blank)
- "Please enter your phone number in the format xxx-xxxx" (it wants three numbers followed by a dash, followed by four numbers)
- "Please enter a valid e-mail address" (the thing you've entered doesn't look like a valid e-mail address)
- "Your password needs to be between 8 and 30 characters long, and contain one uppercase letter, one symbol, and a number" (seriously?)

This is called **form validation** — when you enter data, the web application checks it to see if it is correct. If correct, the application allows the data to be submitted to the server and (usually) saved in a database; if not, it gives you error messages to explain what you've done wrong (provided you've done it right). Form validation can be implemented in a number of different ways.
The truth is that none of us *like* filling out forms — there is a lot of evidence to show that users get annoyed by forms, and it will cause them to leave and go somewhere else if they are done poorly. In short, forms suck.
We want to make filling out web forms as painless as possible. So why do we insist on blocking our users at every turn? There are three main reasons:

- **We want to get the right data, in the right format** — our applications won't work properly if our user's data is stored in any old format they like, or if they don't enter the correct information in the correct places, or omit it altogether.
- **We want to protect our users** — if they enter an easy-to-guess password, or no password at all, malicious users can easily get into their accounts and steal their data.
- **We want to protect ourselves** — there are many ways that malicious users can misuse unprotected forms to damage the application they are part of (see Website security).

Different types of form validation

There are 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. This is more user-friendly than server-side validation as it gives an instant response. This can be further subdivided:
  - JavaScript validation is coded using JavaScript. It is completely customizable.
  - Built-in form validation is done with HTML5 form validation features, and generally doesn't require JavaScript. This has better performance, but it is not as customizable.
- 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 it is put into the database. If the data is wrong, a response is sent back to the client to tell the user what went wrong. Server-side validation is not as user-friendly as client-side validation, as it requires a round trip to the server, but it is your application's last line of defense against bad (meaning incorrect, or even malicious) data. All popular server-side frameworks have features for **validating** and **sanitizing** data (making it safe).

In the real world, developers tend to use a combination of client-side and server-side validation, to be on the safe side.

Using built-in form validation

One of the features of HTML5 is the ability to validate most user data without relying on scripts. This is done by using validation attributes on form elements, which allow you to specify rules for a form input like whether a value needs to be filled in, the minimum and maximum length of the data, whether it needs to

be a number, an email address, or something else, and a pattern that it must match. If the entered data follows all those rules, it is considered valid; if not, it is considered invalid.
When an element is valid:

- The element matches the `:valid` CSS pseudo-class; this will let 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 element matches the `:invalid` CSS pseudo-class; this will let 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 HTML5 features that can be used to validate `<input>` elements.
Let's start with a simple example — an input that allows you to choose your favorite fruit out of a choice of banana or cherry. This involves a simple text `<input>` with a matching label, and a submit `<button>`. You can find the source code on GitHub as fruit-start.html, and a live example below:

Open in CodePenOpen in JSFiddle

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

The required attribute

The simplest HTML5 validation feature to use is the `required` attribute — if you want to make an input mandatory, you can mark the element using 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>
```

Also take note of the CSS included in the example file:

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

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

This causes the input to have a bright red dashed border when it is invalid, and a more subtle black border when valid. Try out the new behaviour in the example below:

Open in CodePenOpen in JSFiddle

Validating against a regular expression

Another very common validation feature is the `pattern` attribute, which expects a Regular Expression as its value. A regular expression (regex) is a pattern that can be used to match character combinations in text strings, so they are ideal for form validation (as well as variety of other uses in JavaScript). Regexs are quite complex and we do not 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, etc.)
- `abc` — matches `a`, followed by `b`, followed by `c`.
- `a*` — matches the character a, zero or more times (`+` matches a character one or more times).
- `[^a]` — matches one character that is **not** a.
- `a|b` — matches one character that is a or b.
- `[abc]` — matches one character that is a, b, or c.
- `[^abc]` — matches one character that is **not** a, b, or c.
- `[a-z]` — matches any character in the range a–z, lower case only (you can use `[A-Za-z]` for lower and upper case, and `[A-Z]` for upper case only).
- `a.c` — matches a, followed by any character, followed by c.
- `a{5}` — matches a, 5 times.
- `a{5,7}` — matches a, 5 to 7 times, but no less or more.
  You can use numbers and other characters in these expressions too, such as:

- `[ -]` — matches a space or a dash.
- `[0-9]` — matches any digit in the range 0 to 9.
  You can combine these in pretty much any way you want, specifying different parts one after the other:

- `[Ll].*k` — A single character that is an upper or lowercase L, followed by zero or more characters of any type, followed by a single lowercase k.
- `[A-Z][A-Za-z' -]+` — A single upper case character followed by one or more characters that are an upper or lower case letter, a dash, an apostrophe, or a space. This could be used to validate the city/town names of English-speaking countries, which need to start with a capital letter, but don't contain any other characters. Examples from the UK include Manchester, Ashton-under-lyne, and Bishop's Stortford.
- `[0-9]{3}[ -][0-9]{3}[ -][0-9]{4}` — A simple match for a US domestic phone number — three digits, followed by a space or a dash, followed by three digits, followed by a space or a dash, followed by four digits. You might have to make this more complex, as some people write their area code in parentheses, but it works for a simple demonstration.
  Anyway, let's implement an example — update your HTML to add a `pattern` attribute, like so:

```
<form>
  <label for="choose">Would you prefer a banana or a cherry?</label>
  <input id="choose" name="i_like" required pattern="banana|cherry">
  <button>Submit</button>
</form>
```

Open in CodePenOpen in JSFiddle

In this example, the `<input>` element accepts one of two possible values: the string "banana" or the string "cherry".
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 you get on! Try to make them fruit-related where possible, so your examples make sense!

Note: Some `<input>` element types do not need a `pattern` attribute to be validated. Specifying the `email` type for example validates the inputted value against a regular expression matching a well-formed email address (or a comma-separated list of email addresses if it has the `multiple` attribute). As a further example, fields with the `url` type automatically require a properly-formed URL.

Note: The `<textarea>` element does not support the `pattern` attribute.

Constraining the length of your entries

All text fields created by (`<input>` or `<textarea>`) can be constrained in size using the `minlength` and `maxlength` attributes. A field is invalid if its value is shorter than the `minlength` value or longer than the `maxlength` value. Browsers often don't let the user type a longer value than expected into text fields anyway, but it is useful to have this fine-grained control available.

For number fields (i.e. `<input type="number">`), the `min` and `max` attributes also provide a validation constraint. If the field's value is lower than the `min` attribute or higher than the `max` attribute, the field 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:

```
<form>
  <div>
    <label for="choose">Would you prefer a banana or a cherry?</label>
    <input 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 6 — the same length as banana and cherry. Entering less characters will show as invalid, and entering more is not possible in most browsers.
- We've also given the `number` field a `min` of 1 and a `max` of 10 — entered numbers outside this range will show as invalid, and you won't be able to use the increment/decrement arrows to move the value outside this range.

Here is the example running live:

Open in CodePenOpen in JSFiddle

Note: `<input type="number">` (and other types, like `range`) can also take a `step`attribute, which specifies what increment the value will go up or down by when the input controls are used (like the up and down number buttons).

Full example

Here is a full example to show off usage of HTML's built-in validation features:

```
<form>
  <p>
    <fieldset>
      <legend>Title<abbr title="This field is mandatory">*</abbr></legend>
      <input type="radio" required name="title" id="r1" value="Mr"><label
for="r1">Mr.</label>
      <input type="radio" required name="title" id="r2" value="Ms"><label
for="r2">Ms.</label>
    </fieldset>
  </p>
```

```
  <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?<abbr title="This field is
mandatory">*</abbr></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>
      <option>Cherry</option>
      <option>Apple</option>
      <option>Strawberry</option>
      <option>Lemon</option>
      <option>Orange</option>
    </datalist>
  </p>
  <p>
    <label for="t2">What's your e-mail?</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>
body {
  font: 1em sans-serif;
  padding: 0;
  margin : 0;
}

form {
  max-width: 200px;
  margin: 0;
  padding: 0 5px;
}

p > label {
  display: block;
}

input[type=text],
input[type=email],
```

```
input[type=number],
textarea,
fieldset {
/* required to properly style form
   elements on WebKit based browsers */
  -webkit-appearance: none;

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

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

  -moz-box-sizing: border-box;
  box-sizing: border-box;
}

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

input:focus:invalid {
  outline: none;
}
```

Open in CodePenOpen in JSFiddle

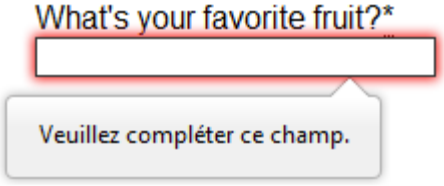
Customized error messages

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

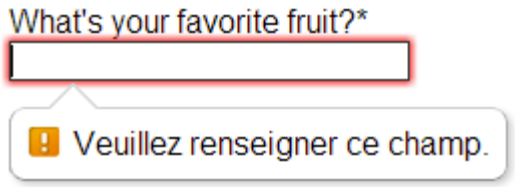
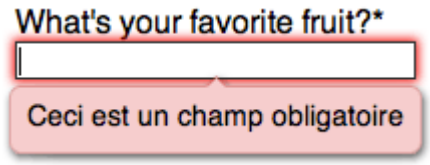
These automated messages have two drawbacks:

- 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)	

French versions of feedback messages on an English page

Browser	Rendering
Chrome 22 (Windows 7)	
Opera 12.10 (Mac OSX)	

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

HTML5 provides the constraint validation API to check and customize the state of a form element. Among other things, it's possible to change the text of the error message. Let's see 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()` 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

Validating forms using JavaScript

If you want to take control over the look and feel of native error messages, or if you want to deal with browsers that do not support HTML's built-in form validation, you must use JavaScript.

The HTML5 constraint validation API

More and more browsers now support the constraint validation API, and it's becoming reliable. This API consists of a set of methods and properties available on each form element.

Constraint validation API properties

Property	Description
`validationMessage`	A localized message describing the validation constraints that the any), or the empty string if the control is not a candidate for constr (`willValidate` is `false`), or the element's value satisfies its constraint
`validity`	A `ValidityState` object describing the validity state of the element.
`validity.customError`	Returns `true` if the element has a custom error; `false` otherwise.
`validity.patternMismatch`	Returns `true` if the element's value doesn't match the provided pat If it returns `true`, the element will match the `:invalid` CSS pseudo-c
`validity.rangeOverflow`	Returns `true` if the element's value is higher than the provided ma If it returns `true`, the element will match the `:invalid` and `:out-of-ran`
`validity.rangeUnderflow`	Returns `true` if the element's value is lower than the provided mini If it returns `true`, the element will match the `:invalid` and `:out-of-ran`
`validity.stepMismatch`	Returns `true` if the element's value doesn't fit the rules provided by otherwise `false` . If it returns `true`, the element will match the `:invalid` and `:out-of-ran`
`validity.tooLong`	Returns `true` if the element's value is longer than the provided ma false If it returns `true`, the element will match the `:invalid` and `:out-of-ran`

Property	Description
`validity.typeMismatch`	Returns true if the element's value is not in the correct syntax; oth If it returns `true`, the element will match the `:invalid` CSS pseudo-c
`validity.valid`	Returns true if the element's value has no validity problems; false If it returns `true`, the element will match the `:valid` CSS pseudo-cla class otherwise.
`validity.valueMissing`	Returns true if the element has no value but is a required field; fal If it returns `true`, the element will match the `:invalid` CSS pseudo-c
`willValidate`	Returns `true` if the element will be validated when the form is subr

Constraint validation API methods

Method	Description
`checkValidity()`	Returns `true` if the element's value has no validity problems; `fals` invalid, this method also causes an `invalid` event at the element
`setCustomValidity(`*`message`*`)`	Adds a custom error message to the element; if you set a custo element is considered to be invalid, and the specified error is dis JavaScript code to establish a validation failure other than those constraint validation API. The message is shown to the user wh If the argument is the empty string, the custom error is cleared.

For legacy browsers, it's possible to use a polyfill such as Hyperform to compensate for the lack of support for the constraint validation API. Since you're already using JavaScript, using a polyfill isn't an added burden to your Web site or Web application's design or implementation.

Example using the constraint validation API

Let's see how to use this API to build custom error messages. First, the HTML:

```
<form novalidate>
```

```
  <p>
    <label for="mail">
      <span>Please enter an email address:</span>
      <input type="email" id="mail" name="mail">
      <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 the CSS pseudo-class `:valid`, `:invalid`, `:in-range` and `:out-of-range` classes. 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.
The `aria-live` attribute makes sure that our custom error message will be presented to everyone, including those using assistive technologies such as screen readers.

CSS

This CSS styles our form and the error output to look more attractive.

```
/* This is just to make the example nicer */
body {
  font: 1em sans-serif;
  padding: 0;
  margin : 0;
}

form {
  max-width: 200px;
}

p * {
  display: block;
}

input[type=email]{
  -webkit-appearance: none;

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

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

  -moz-box-sizing: border-box;
```

```
  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;

  -moz-box-sizing: border-box;
  box-sizing: border-box;
}

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

JavaScript

The following JavaScript code handles the custom error validation.

```
// 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.

var form  = document.getElementsByTagName('form')[0];
var email = document.getElementById('mail');
var error = document.querySelector('.error');

email.addEventListener("input", function (event) {
  // Each time the user types something, we check if the
  // email field is valid.
  if (email.validity.valid) {
    // In case there is an error message visible, if the field
    // is valid, we remove the error message.
    error.innerHTML = ""; // Reset the content of the message
    error.className = "error"; // Reset the visual state of the message
  }
}, false);
form.addEventListener("submit", function (event) {
```

```
  // Each time the user tries to send the data, we check
  // if the email field is valid.
  if (!email.validity.valid) {

    // If the field is not valid, we display a custom
    // error message.
    error.innerHTML = "I expect an e-mail, darling!";
    error.className = "error active";
    // And we prevent the form from being sent by canceling the event
    event.preventDefault();
  }
}, false);
```

Here is the live result:

Open in CodePenOpen in JSFiddle

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 just with HTML and CSS alone.

Validating forms without a built-in API

Sometimes, such as with legacy browsers or custom widgets, you will not be able to (or will not want to) use the constraint validation API. In that case, you're still able to use JavaScript to validate your form. Validating a form is more a question of user interface than real data validation.
To validate a form, you have to 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, etc. It's up to you. Just remember that form data is always text and is always provided to your script as strings.

What should I do if the form does not 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 which 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, there are some useful articles you should read:

- SmashingMagazine: Form-Field Validation: The Errors-Only Approach
- SmashingMagazine: Web Form Validation: Best Practices and Tutorials
- Six Revision: Best Practices for Hints and Validation in Web Forms
- A List Apart: Inline Validation in Web Forms

Example that doesn't use the constraint validation API

In order to illustrate this, let's rebuild the previous example so that it works with legacy browsers:

```
<form>
  <p>
    <label for="mail">
        <span>Please enter an email address:</span>
        <input type="text" class="mail" id="mail" name="mail">
        <span class="error" aria-live="polite"></span>
    </label>
  <p>
  <!-- Some legacy browsers need to have the `type` attribute
       explicitly set to `submit` on the `button`element -->
  <button type="submit">Submit</button>
</form>
```

As you can see, the HTML is almost the same; we just removed the HTML validation features. Note that ARIA is an independent specification that's not specifically related to HTML5.

CSS

Similarly, the CSS doesn't need to change very much; we just turn the `:invalid` CSS pseudo-class into a real class and avoid using the attribute selector that does not work on Internet Explorer 6.

```
/* This is just to make the example nicer */
body {
  font: 1em sans-serif;
  padding: 0;
  margin : 0;
}

form {
  max-width: 200px;
}

p * {
  display: block;
```

```
}

input.mail {
  -webkit-appearance: none;

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

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

  -moz-box-sizing: border-box;
  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;

  -moz-box-sizing: border-box;
  box-sizing: border-box;
}

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

JavaScript

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

```
// There are fewer ways to pick a DOM node with legacy browsers
var form  = document.getElementsByTagName('form')[0];
var email = document.getElementById('mail');
```

```
// The following is a trick to reach the next sibling Element node in the DOM
// This is dangerous because you can easily build an infinite loop.
// In modern browsers, you should prefer using element.nextElementSibling
var error = email;
while ((error = error.nextSibling).nodeType != 1);

// As per the HTML5 Specification
var emailRegExp = /^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-
]+)*$/;

// Many legacy browsers do not support the addEventListener method.
// Here is a simple way to handle this; it's far from the only one.
function addEvent(element, event, callback) {
  var previousEventCallBack = element["on"+event];
  element["on"+event] = function (e) {
    var output = callback(e);

    // A callback that returns `false` stops the callback chain
    // and interrupts the execution of the event callback.
    if (output === false) return false;

    if (typeof previousEventCallBack === 'function') {
      output = previousEventCallBack(e);
      if(output === false) return false;
    }
  }
};

// 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
addEvent(window, "load", function () {
  // 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 e-mail address.
  var test = email.value.length === 0 || emailRegExp.test(email.value);

  email.className = test ? "valid" : "invalid";
});

// This defines what happens when the user types in the field
addEvent(email, "input", function () {
  var test = email.value.length === 0 || emailRegExp.test(email.value);
  if (test) {
    email.className = "valid";
    error.innerHTML = "";
    error.className = "error";
  } else {
    email.className = "invalid";
  }
});

// This defines what happens when the user tries to submit the data
addEvent(form, "submit", function () {
  var test = email.value.length === 0 || emailRegExp.test(email.value);
```

```
  if (!test) {
    email.className = "invalid";
    error.innerHTML = "I expect an e-mail, darling!";
    error.className = "error active";

    // Some legacy browsers do not support the event.preventDefault() method
    return false;
  } else {
    email.className = "valid";
    error.innerHTML = "";
    error.className = "error";
  }
});
```

The result looks like this:

Open in CodePenOpen in JSFiddle

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 it both cross-platform and on any form you might create. There are many libraries available to perform form validation; you shouldn't hesitate to use them. Here are a few examples:

- Standalone library
  - Validate.js
- jQuery plug-in:
  - Validation

Remote validation

In some cases it can be useful to perform some remote validation. This kind of validation is necessary when the data entered by the user is tied to additional data stored on the server side of your application. One use case for this is registration forms, where you ask for a user name. To avoid duplication, it's smarter to perform an AJAX request to check the availability of the user name rather than asking the user to send the data, then send back the form with an error.

Performing such a validation requires taking a few precautions:

- It requires exposing an API and some data publicly; be sure it is not sensitive data.

- Network lag requires performing asynchronous validation. This requires some UI work in order to be sure that the user will not be blocked if the validation is not performed properly.

Sending data is not enough — we also need to make sure that the data users fill out in forms is in the correct format that we need to process it successfully, and that it won't break our applications. We also want to help our users to fill out our forms correctly and not get frustrated when trying to use our apps. Form validation helps us achieve these goals — this article tells you what you need to know.

Prerequisites:	Computer literacy, a reasonable understanding of HTML, CSS, and Java
Objective:	To understand what form validation is, why it's important, and how to im

What is form validation?

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

- "This field is required" (you can't leave this field blank)
- "Please enter your phone number in the format xxx-xxxx" (it wants three numbers followed by a dash, followed by four numbers)
- "Please enter a valid e-mail address" (the thing you've entered doesn't look like a valid e-mail address)
- "Your password needs to be between 8 and 30 characters long, and contain one uppercase letter, one symbol, and a number" (seriously?)

This is called **form validation** — when you enter data, the web application checks it to see if it is correct. If correct, the application allows the data to be submitted to the server and (usually) saved in a database; if not, it gives you error messages to explain what you've done wrong (provided you've done it right). Form validation can be implemented in a number of different ways.

The truth is that none of us *like* filling out forms — there is a lot of evidence to show that users get annoyed by forms, and it will cause them to leave and go somewhere else if they are done poorly. In short, forms suck.
We want to make filling out web forms as painless as possible. So why do we insist on blocking our users at every turn? There are three main reasons:

- **We want to get the right data, in the right format** — our applications won't work properly if our user's data is stored in any old format they like, or if they don't enter the correct information in the correct places, or omit it altogether.
- **We want to protect our users** — if they enter an easy-to-guess password, or no password at all, malicious users can easily get into their accounts and steal their data.
- **We want to protect ourselves** — there are many ways that malicious users can misuse unprotected forms to damage the application they are part of (see Website security).

Different types of form validation

There are 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. This is more user-friendly than server-side validation as it gives an instant response. This can be further subdivided:
  - JavaScript validation is coded using JavaScript. It is completely customizable.
  - Built-in form validation is done with HTML5 form validation features, and generally doesn't require JavaScript. This has better performance, but it is not as customizable.
- 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 it is put into the database. If the data is wrong, a response is sent back to the client to tell the user what went wrong. Server-side validation is not as user-friendly as client-side validation, as it requires a round trip to the server, but it is your application's last line of defense against bad (meaning incorrect, or even malicious) data. All popular server-side frameworks have features for **validating** and **sanitizing** data (making it safe).
  In the real world, developers tend to use a combination of client-side and server-side validation, to be on the safe side.

Using built-in form validation

One of the features of HTML5 is the ability to validate most user data without relying on scripts. This is done by using validation attributes on form elements, which allow you to specify rules for a form input like whether a value needs to be filled in, the minimum and maximum length of the data, whether it needs to be a number, an email address, or something else, and a pattern that it must match. If the entered data follows all those rules, it is considered valid; if not, it is considered invalid.
When an element is valid:

- The element matches the `:valid` CSS pseudo-class; this will let 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 element matches the `:invalid` CSS pseudo-class; this will let 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 HTML5 features that can be used to validate `<input>` elements.
Let's start with a simple example — an input that allows you to choose your favorite fruit out of a choice of banana or cherry. This involves a simple text `<input>` with a matching label, and a submit `<button>`. You can find the source code on GitHub as fruit-start.html, and a live example below:

Open in CodePenOpen in JSFiddle

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

The required attribute

The simplest HTML5 validation feature to use is the `required` attribute — if you want to make an input mandatory, you can mark the element using 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>
```

Also take note of the CSS included in the example file:

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

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

This causes the input to have a bright red dashed border when it is invalid, and a more subtle black border when valid. Try out the new behaviour in the example below:

Open in CodePenOpen in JSFiddle

Validating against a regular expression

Another very common validation feature is the `pattern` attribute, which expects a Regular Expression as its value. A regular expression (regex) is a pattern that can be used to match character combinations in text strings, so they are ideal for form validation (as well as variety of other uses in JavaScript). Regexs are quite complex and we do not 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, etc.)
- `abc` — matches `a`, followed by `b`, followed by `c`.
- `a*` — matches the character a, zero or more times (`+` matches a character one or more times).
- `[^a]` — matches one character that is **not** a.
- `a|b` — matches one character that is a or b.
- `[abc]` — matches one character that is a, b, or c.
- `[^abc]` — matches one character that is **not** a, b, or c.
- `[a-z]` — matches any character in the range a–z, lower case only (you can use `[A-Za-z]` for lower and upper case, and `[A-Z]` for upper case only).
- `a.c` — matches a, followed by any character, followed by c.
- `a{5}` — matches a, 5 times.
- `a{5,7}` — matches a, 5 to 7 times, but no less or more.

You can use numbers and other characters in these expressions too, such as:

- `[ -]` — matches a space or a dash.
- `[0-9]` — matches any digit in the range 0 to 9.
  You can combine these in pretty much any way you want, specifying different parts one after the other:

- `[Ll].*k` — A single character that is an upper or lowercase L, followed by zero or more characters of any type, followed by a single lowercase k.
- `[A-Z][A-Za-z' -]+` — A single upper case character followed by one or more characters that are an upper or lower case letter, a dash, an apostrophe, or a space. This could be used to validate the city/town names of English-speaking countries, which need to start with a capital letter, but don't contain any other characters. Examples from the UK include Manchester, Ashton-under-lyne, and Bishop's Stortford.
- `[0-9]{3}[ -][0-9]{3}[ -][0-9]{4}` — A simple match for a US domestic phone number — three digits, followed by a space or a dash, followed by three digits, followed by a space or a dash, followed by four digits. You might have to make this more complex, as some people write their area code in parentheses, but it works for a simple demonstration.
  Anyway, let's implement an example — update your HTML to add a `pattern` attribute, like so:

```
<form>
  <label for="choose">Would you prefer a banana or a cherry?</label>
  <input id="choose" name="i_like" required pattern="banana|cherry">
  <button>Submit</button>
</form>
```

Open in CodePenOpen in JSFiddle

In this example, the `<input>` element accepts one of two possible values: the string "banana" or the string "cherry".
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 you get on! Try to make them fruit-related where possible, so your examples make sense!

Note: Some `<input>` element types do not need a `pattern` attribute to be validated. Specifying the `email` type for example validates the inputted value against a regular expression matching a well-formed email address (or a comma-separated list of email addresses if it has the `multiple` attribute). As a further example, fields with the `url` type automatically require a properly-formed URL.

Note: The `<textarea>` element does not support the `pattern` attribute.

Constraining the length of your entries

All text fields created by (`<input>` or `<textarea>`) can be constrained in size using the `minlength` and `maxlength` attributes. A field is invalid if its value is shorter than the `minlength` value or longer than the `maxlength` value. Browsers often don't let the user type a longer value than expected into text fields anyway, but it is useful to have this fine-grained control available.
For number fields (i.e. `<input type="number">`), the `min` and `max` attributes also provide a validation constraint. If the field's value is lower than the `min` attribute or higher than the `max` attribute, the field 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:

```
<form>
  <div>
    <label for="choose">Would you prefer a banana or a cherry?</label>
    <input 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 6 — the same length as banana and cherry. Entering less characters will show as invalid, and entering more is not possible in most browsers.
- We've also given the `number` field a `min` of 1 and a `max` of 10 — entered numbers outside this range will show as invalid, and you won't be able to use the increment/decrement arrows to move the value outside this range.

Here is the example running live:

Open in CodePenOpen in JSFiddle

Note: `<input type="number">` (and other types, like `range`) can also take a `step`attribute, which specifies what increment the value will go up or down by when the input controls are used (like the up and down number buttons).

Full example

Here is a full example to show off usage of HTML's built-in validation features:

```
<form>
  <p>
    <fieldset>
```

```
      <legend>Title<abbr title="This field is mandatory">*</abbr></legend>
      <input type="radio" required name="title" id="r1" value="Mr"><label
for="r1">Mr.</label>
      <input type="radio" required name="title" id="r2" value="Ms"><label
for="r2">Ms.</label>
    </fieldset>
  </p>
  <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?<abbr title="This field is
mandatory">*</abbr></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>
      <option>Cherry</option>
      <option>Apple</option>
      <option>Strawberry</option>
      <option>Lemon</option>
      <option>Orange</option>
    </datalist>
  </p>
  <p>
    <label for="t2">What's your e-mail?</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>
body {
  font: 1em sans-serif;
  padding: 0;
  margin : 0;
}

form {
  max-width: 200px;
  margin: 0;
  padding: 0 5px;
}
```

```
p > label {
  display: block;
}

input[type=text],
input[type=email],
input[type=number],
textarea,
fieldset {
/* required to properly style form
   elements on WebKit based browsers */
  -webkit-appearance: none;

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

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

  -moz-box-sizing: border-box;
  box-sizing: border-box;
}

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

input:focus:invalid {
  outline: none;
}
```

Open in CodePenOpen in JSFiddle

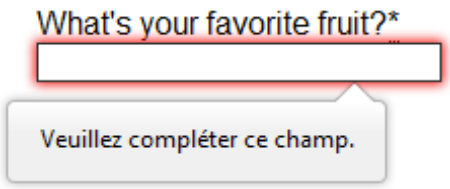
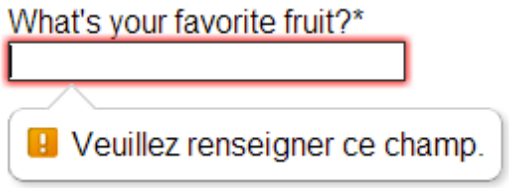
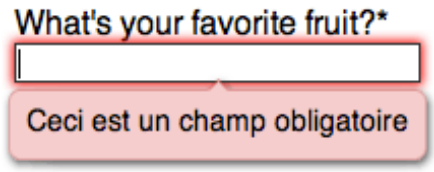
Customized error messages

As seen in the examples above, each time the 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)	
Chrome 22 (Windows 7)	
Opera 12.10 (Mac OSX)	

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

HTML5 provides the constraint validation API to check and customize the state of a form element. Among other things, it's possible to change the text of the error message. Let's see 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()` 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

Validating forms using JavaScript

If you want to take control over the look and feel of native error messages, or if you want to deal with browsers that do not support HTML's built-in form validation, you must use JavaScript.

The HTML5 constraint validation API

More and more browsers now support the constraint validation API, and it's becoming reliable. This API consists of a set of methods and properties available on each form element.

Constraint validation API properties

Property	Description
`validationMessage`	A localized message describing the validation constraints that the any), or the empty string if the control is not a candidate for constr (`willValidate` is `false`), or the element's value satisfies its constraint
`validity`	A `ValidityState` object describing the validity state of the element.
`validity.customError`	Returns `true` if the element has a custom error; `false` otherwise.
`validity.patternMismatch`	Returns `true` if the element's value doesn't match the provided pat If it returns `true`, the element will match the `:invalid` CSS pseudo-c
`validity.rangeOverflow`	Returns `true` if the element's value is higher than the provided max If it returns `true`, the element will match the `:invalid` and `:out-of-ran`
`validity.rangeUnderflow`	Returns `true` if the element's value is lower than the provided mini If it returns `true`, the element will match the `:invalid` and `:out-of-ran`
`validity.stepMismatch`	Returns `true` if the element's value doesn't fit the rules provided by otherwise `false` . If it returns `true`, the element will match the `:invalid` and `:out-of-ran`
`validity.tooLong`	Returns `true` if the element's value is longer than the provided max false

Property	Description
	If it returns `true`, the element will match the `:invalid` and `:out-of-ran`
`validity.typeMismatch`	Returns true if the element's value is not in the correct syntax; oth If it returns `true`, the element will match the `:invalid` CSS pseudo-c
`validity.valid`	Returns true if the element's value has no validity problems; false If it returns `true`, the element will match the `:valid` CSS pseudo-cla class otherwise.
`validity.valueMissing`	Returns true if the element has no value but is a required field; fal If it returns `true`, the element will match the `:invalid` CSS pseudo-c
`willValidate`	Returns `true` if the element will be validated when the form is subr

Constraint validation API methods

Method	Description
`checkValidity()`	Returns `true` if the element's value has no validity problems; `fals` invalid, this method also causes an `invalid` event at the element
`setCustomValidity(`*`message`*`)`	Adds a custom error message to the element; if you set a custo element is considered to be invalid, and the specified error is dis JavaScript code to establish a validation failure other than those constraint validation API. The message is shown to the user wh If the argument is the empty string, the custom error is cleared.

For legacy browsers, it's possible to use a polyfill such as Hyperform to compensate for the lack of support for the constraint validation API. Since you're already using JavaScript, using a polyfill isn't an added burden to your Web site or Web application's design or implementation.

Example using the constraint validation API

Let's see how to use this API to build custom error messages. First, the HTML:

```
<form novalidate>
  <p>
    <label for="mail">
      <span>Please enter an email address:</span>
      <input type="email" id="mail" name="mail">
      <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 the CSS pseudo-class `:valid`, `:invalid`, `:in-range` and `:out-of-range` classes. 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.

The `aria-live` attribute makes sure that our custom error message will be presented to everyone, including those using assistive technologies such as screen readers.

CSS

This CSS styles our form and the error output to look more attractive.

```
/* This is just to make the example nicer */
body {
  font: 1em sans-serif;
  padding: 0;
  margin : 0;
}

form {
  max-width: 200px;
}

p * {
  display: block;
}

input[type=email]{
  -webkit-appearance: none;

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

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

```
  -moz-box-sizing: border-box;
  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;

  -moz-box-sizing: border-box;
  box-sizing: border-box;
}

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

JavaScript

The following JavaScript code handles the custom error validation.

```
// 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.

var form  = document.getElementsByTagName('form')[0];
var email = document.getElementById('mail');
var error = document.querySelector('.error');

email.addEventListener("input", function (event) {
  // Each time the user types something, we check if the
  // email field is valid.
  if (email.validity.valid) {
    // In case there is an error message visible, if the field
    // is valid, we remove the error message.
    error.innerHTML = ""; // Reset the content of the message
    error.className = "error"; // Reset the visual state of the message
  }
}, false);
```

```
form.addEventListener("submit", function (event) {
  // Each time the user tries to send the data, we check
  // if the email field is valid.
  if (!email.validity.valid) {

    // If the field is not valid, we display a custom
    // error message.
    error.innerHTML = "I expect an e-mail, darling!";
    error.className = "error active";
    // And we prevent the form from being sent by canceling the event
    event.preventDefault();
  }
}, false);
```

Here is the live result:

Open in CodePenOpen in JSFiddle

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 just with HTML and CSS alone.

Validating forms without a built-in API

Sometimes, such as with legacy browsers or custom widgets, you will not be able to (or will not want to) use the constraint validation API. In that case, you're still able to use JavaScript to validate your form. Validating a form is more a question of user interface than real data validation.
To validate a form, you have to 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, etc. It's up to you. Just remember that form data is always text and is always provided to your script as strings.

What should I do if the form does not 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 which 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, there are some useful articles you should read:

- SmashingMagazine: Form-Field Validation: The Errors-Only Approach
- SmashingMagazine: Web Form Validation: Best Practices and Tutorials
- Six Revision: Best Practices for Hints and Validation in Web Forms
- A List Apart: Inline Validation in Web Forms

Example that doesn't use the constraint validation API

In order to illustrate this, let's rebuild the previous example so that it works with legacy browsers:

```
<form>
  <p>
    <label for="mail">
        <span>Please enter an email address:</span>
        <input type="text" class="mail" id="mail" name="mail">
        <span class="error" aria-live="polite"></span>
    </label>
  <p>
  <!-- Some legacy browsers need to have the `type` attribute
       explicitly set to `submit` on the `button`element -->
  <button type="submit">Submit</button>
</form>
```

As you can see, the HTML is almost the same; we just removed the HTML validation features. Note that ARIA is an independent specification that's not specifically related to HTML5.

CSS

Similarly, the CSS doesn't need to change very much; we just turn the `:invalid` CSS pseudo-class into a real class and avoid using the attribute selector that does not work on Internet Explorer 6.

```
/* This is just to make the example nicer */
body {
  font: 1em sans-serif;
  padding: 0;
  margin : 0;
}

form {
  max-width: 200px;
}

p * {
  display: block;
```

```
}

input.mail {
  -webkit-appearance: none;

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

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

  -moz-box-sizing: border-box;
  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;

  -moz-box-sizing: border-box;
  box-sizing: border-box;
}

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

JavaScript

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

```
// There are fewer ways to pick a DOM node with legacy browsers
var form  = document.getElementsByTagName('form')[0];
var email = document.getElementById('mail');
```

```
// The following is a trick to reach the next sibling Element node in the DOM
// This is dangerous because you can easily build an infinite loop.
// In modern browsers, you should prefer using element.nextElementSibling
var error = email;
while ((error = error.nextSibling).nodeType != 1);

// As per the HTML5 Specification
var emailRegExp = /^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-
]+)*$/;

// Many legacy browsers do not support the addEventListener method.
// Here is a simple way to handle this; it's far from the only one.
function addEvent(element, event, callback) {
  var previousEventCallBack = element["on"+event];
  element["on"+event] = function (e) {
    var output = callback(e);

    // A callback that returns `false` stops the callback chain
    // and interrupts the execution of the event callback.
    if (output === false) return false;

    if (typeof previousEventCallBack === 'function') {
      output = previousEventCallBack(e);
      if(output === false) return false;
    }
  }
};

// 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
addEvent(window, "load", function () {
  // 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 e-mail address.
  var test = email.value.length === 0 || emailRegExp.test(email.value);

  email.className = test ? "valid" : "invalid";
});

// This defines what happens when the user types in the field
addEvent(email, "input", function () {
  var test = email.value.length === 0 || emailRegExp.test(email.value);
  if (test) {
    email.className = "valid";
    error.innerHTML = "";
    error.className = "error";
  } else {
    email.className = "invalid";
  }
});

// This defines what happens when the user tries to submit the data
addEvent(form, "submit", function () {
  var test = email.value.length === 0 || emailRegExp.test(email.value);
```

```
  if (!test) {
    email.className = "invalid";
    error.innerHTML = "I expect an e-mail, darling!";
    error.className = "error active";

    // Some legacy browsers do not support the event.preventDefault() method
    return false;
  } else {
    email.className = "valid";
    error.innerHTML = "";
    error.className = "error";
  }
});
```

The result looks like this:

Open in CodePenOpen in JSFiddle

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 it both cross-platform and on any form you might create. There are many libraries available to perform form validation; you shouldn't hesitate to use them. Here are a few examples:

- Standalone library
  - Validate.js
- jQuery plug-in:
  - Validation

Remote validation

In some cases it can be useful to perform some remote validation. This kind of validation is necessary when the data entered by the user is tied to additional data stored on the server side of your application. One use case for this is registration forms, where you ask for a user name. To avoid duplication, it's smarter to perform an AJAX request to check the availability of the user name rather than asking the user to send the data, then send back the form with an error.

Performing such a validation requires taking a few precautions:

- It requires exposing an API and some data publicly; be sure it is not sensitive data.

- Network lag requires performing asynchronous validation. This requires some UI work in order to be sure that the user will not be blocked if the validation is not performed properly.

As in the previous article, HTML forms can send an HTTP request declaratively. But forms can also prepare an HTTP request to send via JavaScript. This article explores ways to do that.

A form is not always a form

With open Web apps, it's increasingly common to use HTML forms other than literal forms for humans to fill out — more and more developers are taking control over transmitting data.

Gaining control of the global interface

Standard HTML form submission loads the URL where the data was sent, which means the browser window navigates with a full page load. Avoiding a full page load can provide a smoother experience by hiding flickering and network lag.

Many modern UIs only use HTML forms to collect input from the user. When the user tries to send the data, the application takes control and transmits the data asynchronously in the background, updating only the parts of the UI that require changes.

Sending arbitrary data asynchronously is known as AJAX, which stands for "Asynchronous JavaScript And XML."

How is it different?

AJAX uses the `XMLHttpRequest` (XHR) DOM object. It can build HTTP requests, send them, and retrieve their results.

Note: Older AJAX techniques might not rely on `XMLHttpRequest`. For example, JSONP combined with the `eval()` function. It works, but it's not recommended because of serious security issues. The only reason to use this is for legacy browsers that lack support for `XMLHttpRequest` or JSON, but those are very old browsers indeed! **Avoid such techniques.**

Historically, `XMLHttpRequest` was designed to fetch and send XML as an exchange format. However, JSON superseded XML and is overwhelmingly more common today.

But neither XML nor JSON fit into form data request encoding. Form data (`application/x-www-form-urlencoded`) is made of URL-encoded lists of key/value pairs. For transmitting binary data, the HTTP request is reshaped into `multipart/form-data`.
If you control the front-end (the code that's executed in the browser) and the back-end (the code which is executed on the server), you can send JSON/XML and process them however you want.

But if you want to use a third party service, it's not that easy. Some services only accept form data. There are also cases where it's simpler to use form data. If the data is key/value pairs, or raw binary data, existing back-end tools can handle it with no extra code required.

So how to send such data?

Sending form data

There are 3 ways to send form data, from legacy techniques to the newer `FormData`object. Let's look at them in detail.

Building an XMLHttpRequest manually

`XMLHttpRequest` is the safest and most reliable way to make HTTP requests. To send form data with `XMLHttpRequest`, prepare the data by URL-encoding it, and obey the specifics of form data requests.

Note: To learn more about `XMLHttpRequest`, these articles may interest you: An introductory article to AJAX and a more advanced tutorial about using `XMLHttpRequest`.

Let's rebuild our previous example:

```
<button type="button" onclick="sendData({test:'ok'})">Click Me!</button>
```

As you can see, the HTML hasn't really changed. However, the JavaScript is completely different:

```
function sendData(data) {
  var XHR = new XMLHttpRequest();
  var urlEncodedData = "";
  var urlEncodedDataPairs = [];
```

```
  var name;

  // Turn the data object into an array of URL-encoded key/value pairs.
  for(name in data) {
    urlEncodedDataPairs.push(encodeURIComponent(name) + '=' +
encodeURIComponent(data[name]));
  }

  // Combine the pairs into a single string and replace all %-encoded spaces to
  // the '+' character; matches the behaviour of browser form submissions.
  urlEncodedData = urlEncodedDataPairs.join('&').replace(/%20/g, '+');

  // Define what happens on successful data submission
  XHR.addEventListener('load', function(event) {
    alert('Yeah! Data sent and response loaded.');
  });

  // Define what happens in case of error
  XHR.addEventListener('error', function(event) {
    alert('Oops! Something goes wrong.');
  });

  // Set up our request
  XHR.open('POST', 'https://example.com/cors.php');

  // Add the required HTTP header for form data POST requests
  XHR.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');

  // Finally, send our data.
  XHR.send(urlEncodedData);
}
```

Here's the live result:

Open in CodePenOpen in JSFiddle

Note: This use of `XMLHttpRequest` is subject to the same origin policy if you want to send data to a third party web site. For cross-origin requests, you'll need CORS and HTTP access control.

Using XMLHttpRequest and the FormData object

Building an HTTP request by hand can be overwhelming. Fortunately, a recent XMLHttpRequest specification provides a convenient and simpler way to handle form data requests with the `FormData` object.

The `FormData` object can be used to build form data for transmission, or to get the data within a form element to manage how it's sent. Note that `FormData` objects are "write only", which means you can change them, but not retrieve their contents.

Using this object is detailed in Using FormData Objects, but here are two examples:

Using a standalone FormData object

```
<button type="button" onclick="sendData({test:'ok'})">Click Me!</button>
```

You should be familiar with that HTML sample.

```
function sendData(data) {
  var XHR = new XMLHttpRequest();
  var FD  = new FormData();

  // Push our data into our FormData object
  for(name in data) {
    FD.append(name, data[name]);
  }

  // Define what happens on successful data submission
  XHR.addEventListener('load', function(event) {
    alert('Yeah! Data sent and response loaded.');
  });

  // Define what happens in case of error
  XHR.addEventListener('error', function(event) {
    alert('Oops! Something went wrong.');
  });

  // Set up our request
  XHR.open('POST', 'https://example.com/cors.php');

  // Send our FormData object; HTTP headers are set automatically
  XHR.send(FD);
}
```

Here's the live result:

Open in CodePenOpen in JSFiddle

Using FormData bound to a form element

You can also bind a `FormData` object to a `<form>` element. This creates a `FormData` that represents the data contained in the form.
The HTML is typical:

```
<form id="myForm">
  <label for="myName">Send me your name:</label>
  <input id="myName" name="name" value="John">
  <input type="submit" value="Send Me!">
```

```
</form>
```

But JavaScript takes over the form:

```
window.addEventListener("load", function () {
  function sendData() {
    var XHR = new XMLHttpRequest();

    // Bind the FormData object and the form element
    var FD = new FormData(form);

    // Define what happens on successful data submission
    XHR.addEventListener("load", function(event) {
      alert(event.target.responseText);
    });

    // Define what happens in case of error
    XHR.addEventListener("error", function(event) {
      alert('Oops! Something went wrong.');
    });

    // Set up our request
    XHR.open("POST", "https://example.com/cors.php");

    // The data sent is what the user provided in the form
    XHR.send(FD);
  }

  // Access the form element...
  var form = document.getElementById("myForm");

  // ...and take over its submit event.
  form.addEventListener("submit", function (event) {
    event.preventDefault();

    sendData();
  });
});
```

Here's the live result:

Open in CodePenOpen in JSFiddle

You can even get more involved with the process by using the form's `elements` property to get a list of all of the data elements in the form and manually managing them one at a time. To learn more about that, see the example in the Accessing the element list's contents in HTMLFormElement.elements.

Building a DOM in a hidden iframe

The oldest way to asynchronously send form data is building a form with the DOM API, then sending its data into a hidden `<iframe>`. To access the result of your submission, retrieve the content of the `<iframe>`.

Warning: Avoid using this technique. It's a security risk with third-party services because it leaves you open to script injection attacks. If you use HTTPS, it can affect the same origin policy, which can render the content of an `<iframe>`unreachable. However, this method may be your only option if you need to support very old browsers.

Here is an example:

```
<button onclick="sendData({test:'ok'})">Click Me!</button>
// Create the iFrame used to send our data
var iframe = document.createElement("iframe");
iframe.name = "myTarget";

// Next, attach the iFrame to the main document
window.addEventListener("load", function () {
  iframe.style.display = "none";
  document.body.appendChild(iframe);
});

// This is the function used to actually send the data
// It takes one parameter, which is an object populated with key/value pairs.
function sendData(data) {
  var name,
      form = document.createElement("form"),
      node = document.createElement("input");

  // Define what happens when the response loads
  iframe.addEventListener("load", function () {
    alert("Yeah! Data sent.");
  });

  form.action = "http://www.cs.tut.fi/cgi-bin/run/~jkorpela/echo.cgi";
  form.target = iframe.name;

  for(name in data) {
    node.name  = name;
    node.value = data[name].toString();
    form.appendChild(node.cloneNode());
  }

  // To be sent, the form needs to be attached to the main document.
  form.style.display = "none";
  document.body.appendChild(form);

  form.submit();

  // Once the form is sent, remove it.
  document.body.removeChild(form);
}
```

Here's the live result:

Open in CodePenOpen in JSFiddle

Dealing with binary data

If you use a `FormData` object with a form that includes `<input type="file">` widgets, the data will be processed automatically. But to send binary data by hand, there's extra work to do.
There are many sources for binary data on the modern Web: `FileReader`, `Canvas`, and WebRTC, for example. Unfortunately, some legacy browsers can't access binary data or require complicated workarounds. Those legacy cases are out of this article's scope. If you want to know more about the `FileReader` API, read Using files from web applications.
Sending binary data with support for `FormData` is straightfoward. Use the `append()`method and you're done. If you have to do it by hand, it's trickier. In the following example, we use the `FileReader` API to access binary data and then build the multi-part form data request by hand:

```
<form id="myForm">
  <p>
    <label for="i1">text data:</label>
    <input id="i1" name="myText" value="Some text data">
  </p>
  <p>
    <label for="i2">file data:</label>
    <input id="i2" name="myFile" type="file">
  </p>
  <button>Send Me!</button>
</form>
```

As you see, the HTML is a standard `<form>`. There's nothing magical going on. The "magic" is in the JavaScript:

```
// Because we want to access DOM node,
// we initialize our script at page load.
window.addEventListener('load', function () {

  // These variables are used to store the form data
  var text = document.getElementById("i1");
```

```
var file = {
      dom    : document.getElementById("i2"),
      binary : null
    };

// Use the FileReader API to access file content
var reader = new FileReader();

// Because FileReader is asynchronous, store its
// result when it finishes to read the file
reader.addEventListener("load", function () {
  file.binary = reader.result;
});

// At page load, if a file is already selected, read it.
if(file.dom.files[0]) {
  reader.readAsBinaryString(file.dom.files[0]);
}

// If not, read the file once the user selects it.
file.dom.addEventListener("change", function () {
  if(reader.readyState === FileReader.LOADING) {
    reader.abort();
  }

  reader.readAsBinaryString(file.dom.files[0]);
});

// sendData is our main function
function sendData() {
  // If there is a selected file, wait it is read
  // If there is not, delay the execution of the function
  if(!file.binary && file.dom.files.length > 0) {
    setTimeout(sendData, 10);
    return;
  }

  // To construct our multipart form data request,
  // We need an XMLHttpRequest instance
  var XHR = new XMLHttpRequest();

  // We need a separator to define each part of the request
  var boundary = "blob";

  // Store our body request in a string.
  var data = "";

  // So, if the user has selected a file
  if (file.dom.files[0]) {
    // Start a new part in our body's request
    data += "--" + boundary + "\r\n";

    // Describe it as form data
    data += 'content-disposition: form-data; '
```

```
      // Define the name of the form data
            + 'name="'         + file.dom.name          + '"; '
      // Provide the real name of the file
            + 'filename="'     + file.dom.files[0].name + '"\r\n';
      // And the MIME type of the file
      data += 'Content-Type: ' + file.dom.files[0].type + '\r\n';

      // There's a blank line between the metadata and the data
      data += '\r\n';

      // Append the binary data to our body's request
      data += file.binary + '\r\n';
    }

    // Text data is simpler
    // Start a new part in our body's request
    data += "--" + boundary + "\r\n";

    // Say it's form data, and name it
    data += 'content-disposition: form-data; name="' + text.name + '"\r\n';
    // There's a blank line between the metadata and the data
    data += '\r\n';

    // Append the text data to our body's request
    data += text.value + "\r\n";

    // Once we are done, "close" the body's request
    data += "--" + boundary + "--";

    // Define what happens on successful data submission
    XHR.addEventListener('load', function(event) {
      alert('Yeah! Data sent and response loaded.');
    });

    // Define what happens in case of error
    XHR.addEventListener('error', function(event) {
      alert('Oops! Something went wrong.');
    });

    // Set up our request
    XHR.open('POST', 'https://example.com/cors.php');

    // Add the required HTTP header to handle a multipart form data POST request
    XHR.setRequestHeader('Content-Type','multipart/form-data; boundary=' + boundary);

    // And finally, send our data.
    XHR.send(data);
  }

  // Access our form...
  var form = document.getElementById("myForm");

  // ...to take over the submit event
  form.addEventListener('submit', function (event) {
```

```
    event.preventDefault();
    sendData();
  });
});
```

Here's the live result:

Open in CodePenOpen in JSFiddle

Conclusion

All web developers learn very quickly (and sometimes painfully) that the Web is a very rough place for them. Our worst curse is legacy browsers. Okay, let's admit it, when we said "legacy browser" we all have in mind old versions of Internet Explorer ... but it's far from the only one. A one-year-old Firefox such as the ESR version is a legacy browser too. And in the mobile world? When neither the browser nor the OS can be updated? Yes, there are many older Android phones or iPhones that have stock browsers that are not up to date. These are also legacy browsers.

Sadly, dealing with that wilderness is part of the job. Fortunately, there are a few tricks to know that can help you to solve about 80% of the problems caused by legacy browsers.

Learn about the issues

Actually, the most important thing is to read documentation about those browsers to try to understand the common patterns. For example, CSS support is the biggest issue with HTML forms in many cases. You are at the right place to start. Just check the support of the elements (or DOM interface) you want to use. MDN has compatibility tables available for many elements, properties or APIs that can be used in a web page. But there are other resources that can be amazingly helpful:

Browser vendor documentation

- Mozilla: You're in the right place, just browse MDN
- Microsoft: Internet Explorer Standards Support Documentation
- WebKit: Because there are several different editions of this engine, things are a little trickier.
  - The WebKit blog and Planet WebKit aggregate the best articles by WebKit core developers.
  - Chrome platform status site is also important.
  - As well as the Apple web site.

Independent documentation

- Can I Use has information about support for cutting edge technologies.
- Quirks Mode is an amazing resource about browsers' compatibility. The mobile part is one of the best available at the moment.
- Position Is Everything is the best resource available about rendering bugs in legacy browsers and their work-arounds (if any).
- Mobile HTML5 has compatibility information for a wide range of mobile browsers, not just the "top 5" (including Nokia, Amazon, and Blackberry).

Make things simple

Because HTML forms involves complex interaction, there is one rule of thumb: keep it as simple as possible. There are so many cases where we want forms that are "nicer" or "with advanced functionality", but building efficient HTML Forms is not a question of design or technology. Just as a reminder, take the time to read this article about forms usability on UX For The Masses.

Graceful degradation is web developer's best friend

Graceful degradation and progressive enhancement are development patterns that allow you to build great stuff by supporting a wide range of browsers at the same time. When you build something for a modern browser, and you want to be sure it will work, one way or another, on legacy browsers, you are performing graceful degradation.
Let's see some examples related to HTML forms.

HTML input types

The new input types brought by HTML5 are very cool because the way they degrade is highly predictable. If a browser does not know the value of the `type` attribute of an `<input>` element, it will fall back as if the value were `text`.

```
<label for="myColor">
  Pick a color
  <input type="color" id="myColor" name="color">
</label>
```

Chrome 24	Firefox 18
Pick a color 	Pick a color

CSS Attribute Selectors

The CSS Attribute selectors are very useful with HTML Forms but some legacy browsers do not support it. In that case, it's customary to double the type with an equivalent class:

```
<input type="number" class="number">
input[type=number] {
  /* This can fail in some browsers */
}

input.number {
  /* This will work everywhere */
}
```

Note that the following is useless (because it's redundant) and can fail in some browsers:

```
input[type=number],
input.number {
  /* This can fail in some browsers because if they do not understand
     one of the selectors, they will skip the whole rule */
}
```

Form buttons

There are two ways to define buttons within HTML forms:

- The `<input>` element with its attribute `type` set to the values `button`, `submit`, `reset`or `image`

- The `<button>` element
  The `<input>` element can make things a little difficult if you want to apply some CSS by using the element selector:

```
<input type="button" class="button" value="click me">
input {
  /* This rule turns off the default rendering for buttons defined with an input
element */
  border: 1px solid #CCC;
}

input.button {
  /* This does NOT restore the default rendering */
  border: none;
}

input.button {
  /* That doesn't either! Actually there is no standard way to do it in any browser
*/
  border: auto;
}
```

The `<button>` element suffers from two possible issues:

- A bug in some old versions of Internet Explorer. When the user click the button, it's not the content of the `value` attribute that is sent, but the HTML content available between the starting and ending tag of the `<button>` element. This is an issue only if you want to send such a value, for example if the processing of the data depends on which button the user clicks.
- Some very old browsers does not use `submit` as the default value for the `type`attribute, so it's recommended to always set the attribute `type` on `<button>`elements.

```
<!-- Clicking this button sends "<em>Do A</em>" instead of "A" in some cases -->
<button type="submit" name="IWantTo" value="A">
  <em>Do A</em>
</button>
```

Choosing one solution or the other is up to you based on your project's constraints.

Let go of CSS

The biggest issue with HTML Forms and legacy browsers is the support for CSS. As you can see from the complexity of the Property compatibility table for form widgets article, it's very difficult. Even if it's still possible to do a few adjustments on text elements (such as sizing or font color), there are always side effects. The best approach remains to not style HTML Form widgets at all. But you can still apply styles to all the surrounding items. If you are a

professional and if your client requires it, in that case, you can investigate some hard techniques such as rebuilding widgets with JavaScript. But in that case, do not hesitate to charge your client for such foolishness.

Feature detection and polyfills

While JavaScript is an awesome technology in modern browsers, legacy browsers have many issues with it.

Unobtrusive JavaScript

One of the biggest problems is the availability of APIs. For that reason, it's considered best practice to work with "unobtrusive" JavaScript. It's a development pattern that defines two requirements:

- A strict separation between structure and behaviors.
- If the code breaks, the content and the basic functionalities must remain accessible and usable.

The principles of unobtrusive JavaScript (originally written by Peter-Paul Koch for Dev.Opera.com and now moved to Docs.WebPlatform.org) describes these ideas very well.

The Modernizr library

There are many cases where a good "polyfill" can help a lot by providing a missing API. A polyfill is a bit of JavaScript that "fills in the holes" in the functionality of legacy browsers. While they can be used to improve support for any functionality, using them for JavaScript is less risky than for CSS or HTML; there many cases where JavaScript can break (network issues, script conflicts, etc.). But for JavaScript, if you work with unobstructive JavaScript in mind, if polyfills are missing, it's no big deal.

The best way to polyfill missing API is by using the Modernizr library and its spin-off project: YepNope. Modernizr is a library that allows you to test the availability of functionality in order to act accordingly. YepNope is a conditional loading library.

Here is an example:

```
Modernizr.load({
```

```
  // This tests if your browser supports the HTML5 form validation API
  test : Modernizr.formvalidation,

  // If the browser does not support it, the following polyfill is loaded
  nope : form-validation-API-polyfill.js,

  // In any case, your core App file that depends on that API is loaded
  both : app.js,

  // Once both files are loaded, this function is called in order to initialize the
App.
  complete : function () {
    app.init();
  }
});
```

The Modernizr team conveniently maintains a list of great polyfills. Just pick what you need.

Note: Modernizr has other awesome features to help you in dealing with unobstructive JavaScript and graceful degradation techniques. Please read the Modernizr documentation.

Pay attention to performance

Even though scripts like Modernizr are very aware of performance, loading a 200 kilobyte polyfill can affect the performance of your application. This is especially critical with legacy browsers; many of them have a very slow JavaScript engine that can make the execution of all your polyfills painful for the user. Performance is a subject on its own, but legacy browsers are very sensitive to it: basically, they are slow and the more polyfills they need, the more JavaScript they have to process. So they are doubly burdened compared to modern browsers. Test your code with legacy browsers to see how they actually perform. Sometimes, dropping some functionality leads to a better user experience than having exactly the same functionality in all browsers. As a last reminder, just always think about the end users.

In this article, the user will learn how to use CSS with HTML forms to make them (hopefully) more beautiful. Surprisingly, this can be a little bit tricky. For historical and technical reasons, form widgets don't mesh well with CSS. Because of those difficulties, many developers choose to build their own HTML widgets to gain control over their look and feel. However, with modern browsers, web designers have more and more control over the design of form elements. Let's dig in.

Why is it so hard to style form widgets with CSS?

In the early days of the Web—around 1995—form controls were added to HTML in the HTML 2 specification. Due to the complexity of form widgets, implementors chose to rely on the underlying operating system to manage and render them.
A few years later CSS was created, and what was a technical necessity, that is, using native widgets to implement form controls, became a style requirement. In the early days of CSS, styling form controls wasn't a priority.

Because users are accustomed to the visual appearance of their respective platforms, browser vendors are reluctant to make form controls stylable; and to this day it is still extremely difficult to rebuild all the controls to make them stylable.

Even today, not a single browser fully implements CSS 2.1. Over time, however, browser vendors have improved their support of CSS for form elements, and even though there's a bad reputation for its usability, you can now use CSS to style HTML forms.

Not all widgets are created equal when CSS is involved

At present, some difficulties remain when using CSS with forms. These problems can be divided in three categories:

The good

Some elements can be styled with few if any problems across platforms. These include the following structural elements:

1. `<form>`
2. `<fieldset>`
3. `<label>`
4. `<output>`

This also includes all text field widgets (both single-line and multi-line), and buttons.

The bad

Some elements can rarely be styled, and may require some complicated tricks, occasionally requiring advanced knowledge of CSS3.

These include the `<legend>` element, but this cannot be positioned properly across all platforms. Checkboxes and radio buttons also can't be styled

directly, however, thanks to CSS3 you can work around this. `placeholder` content is not stylable in any standard way, however, all browsers that implement it also implement proprietary CSS pseudo-elements, or pseudo-classes that let you style it.
We describe how to handle these more specific cases in the article Advanced styling for HTML forms.

The ugly

Some elements simply can't be styled using CSS. These include: all advanced user interface widgets, such as range, color, or date controls; and all the dropdown widgets,
including `<select>`, `<option>`, `<optgroup>` and `<datalist>` elements. The file picker widget is also known not to be stylable at all. The
new `<progress>` and `<meter>` elements also fall in this category.
The main issue with all these widgets, comes from the fact that they have a very complex structure, and CSS is not currently expressive enough to style all the subtle parts of those widgets. If you want to customize those widgets, you have to rely on JavaScript to build a DOM tree you'll be able to style. We explore how to do this in the article How to build custom form widgets.

Basic styling

To style elements that are easy to style with CSS, you shouldn't face any difficulties, since they mostly behave like any other HTML element. However, the user-agent style sheet of every browser can be a little inconsistent, so there are a few tricks that can help you style them in an easier way.

Search fields

Search boxes are the only kind of text fields that can be a little tricky to style. On WebKit based browsers (Chrome, Safari, etc.), you'll have to tweak it with the `-webkit-appearance` proprietary property. We discuss this property further in the article: Advanced styling for HTML forms.

Example

```
<form>
  <input type="search">
</form>
input[type=search] {
  border: 1px dotted #999;
  border-radius: 0;

  -webkit-appearance: none;
}
```

Chrome 25 / Mac OSX

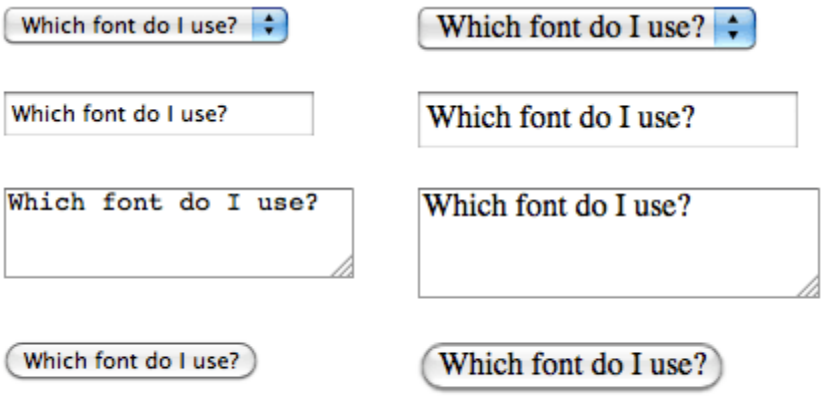
As you can see on this screenshot of the search field on Chrome, the two fields have a border set as in our example. The first field is rendered without using the `-webkit-appearance` property, whereas the second is rendered using `-webkit-appearance:none`. This difference is noticeable.

Fonts and text

CSS font and text features can be used easily with any widget (and yes, you can use `@font-face` with form widgets). However, browsers' behaviors are often inconsistent. By default, some widgets do not inherit `font-family` and `font-size` from their parents. Many browsers use the system default appearance instead. To make your forms' appearance consistent with the rest of your content, you can add the following rules to your stylesheet:

```
button, input, select, textarea {
  font-family : inherit;
  font-size   : 100%;
}
```

The screenshot below shows the difference; on the left is the default rendering of the element in Firefox on Mac OS X, with the platform's default font style in use. On the right are the same elements, with our font harmonization style rules applied.



Firefox 16 / Mac OSX

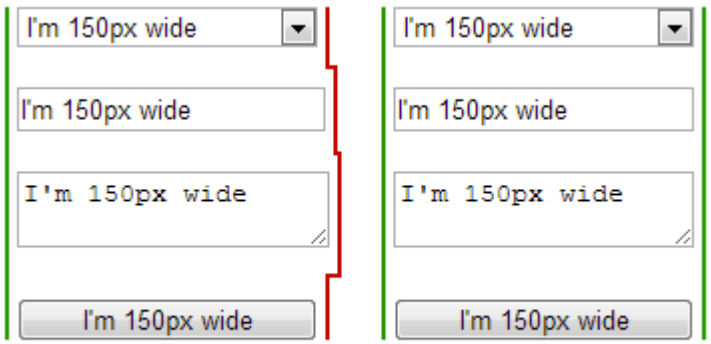
There's a lot of debate as to whether forms look better using the system default styles, or customized styles designed to match your content. This decision is yours to make, as the designer of your site, or Web application.

Box model

All text fields have complete support for every property related to the CSS box model (`width`, `height`, `padding`, `margin`, and `border`). As before, however, browsers rely on the system default styles when displaying these widgets. It's up to you to define how you wish to blend them into your content. If you want to keep the native look and feel of the widgets, you'll face a little difficulty if you want to give them a consistent size.

This is because each widget has their own rules for border, padding and margin. So if you want to give the same size to several different widgets, you have to use the `box-sizing` property:

```
input, textarea, select, button {
  width : 150px;
  margin: 0;

  -webkit-box-sizing: border-box; /* For legacy WebKit based browsers */
     -moz-box-sizing: border-box; /* For legacy (Firefox <29) Gecko based browsers */
          box-sizing: border-box;
}
```



Chrome 22 / Windows 7

In the screenshot above, the left column is built without `box-sizing`, while the right column uses this property with the value `border-box`. Notice how this lets us ensure that all of the elements occupy the same amount of space, despite the platform's default rules for each kind of widget.

Positioning

Positioning of HTML form widgets is generally not a problem; however, there are two elements you should take special note of:

legend

The `<legend>` element is okay to style, except for positioning. In every browser, the `<legend>` element is positioned on top of the top border of its `<fieldset>` parent. There is absolutely no way to change it to be positioned within the HTML flow, away from the top border. You can, however, position it absolutely or relatively, using the `position`property. But otherwise it is part of the fieldset border.

Because the `<legend>` element is very important for accessibility reasons, it will be spoken by assistive technologies as part of the label of each form element inside the fieldset, it's quite often paired with a title, and then hidden in an accessible way. For example:

HTML

```
<fieldset>
  <legend>Hi!</legend>
  <h1>Hello</h1>
</fieldset>
```

CSS

```
legend {
  width: 1px;
  height: 1px;
```

```
  overflow: hidden;
}
```

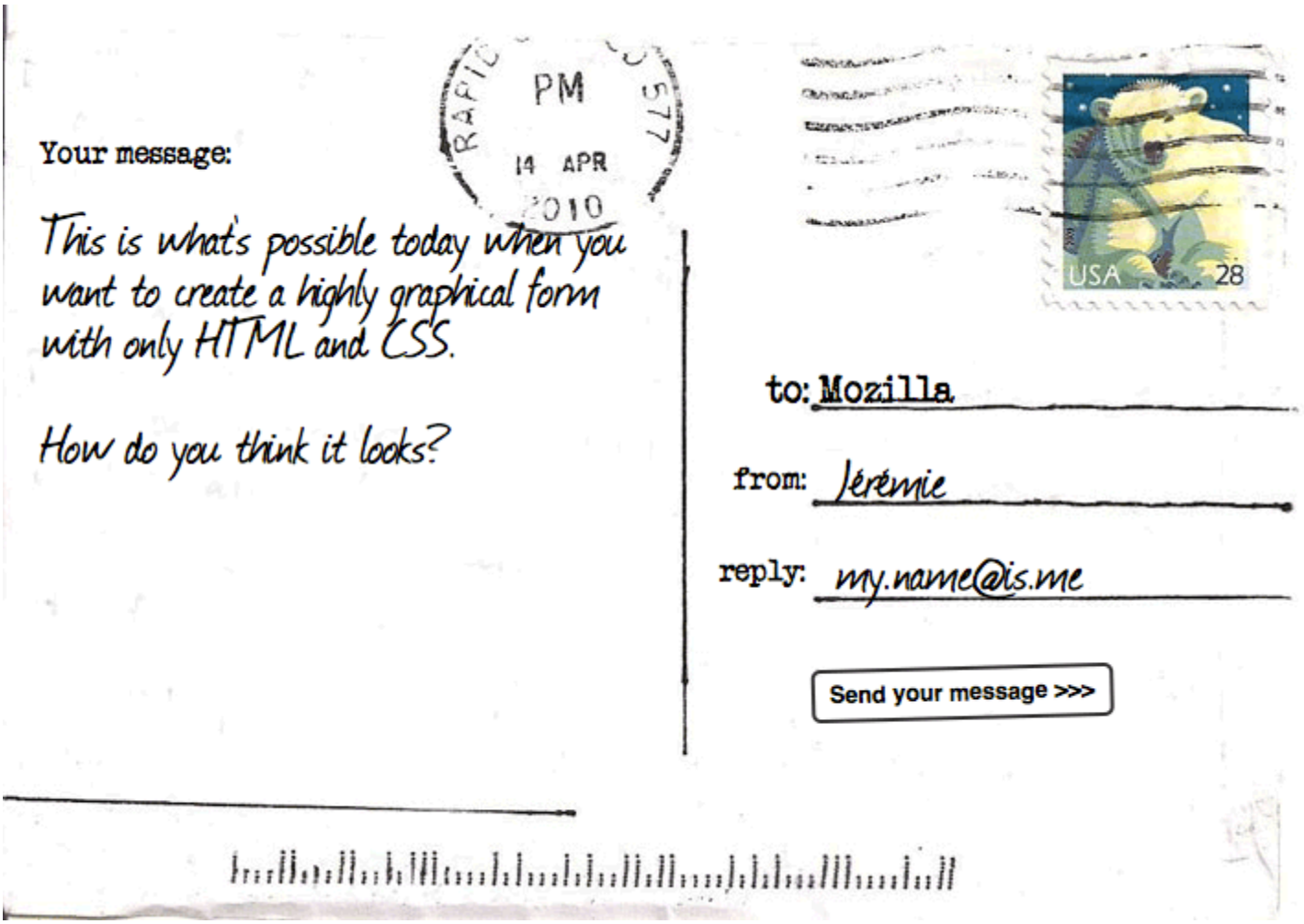
textarea

By default, all browsers consider the `<textarea>` element to be an inline block, aligned to the text bottom line. This is rarely what we actually want to see. To change from `inline-block` to `block`, it's pretty easy to use the `display` property. But if you want to use it inline, it's common to change the vertical alignment:

```
textarea {
  vertical-align: top;
}
```

Example

Let's look at a concrete example of how to style an HTML form. This will help make a lot of these ideas clearer. We will build the following "postcard" contact form:



If you want to follow along with this example, make a local copy of our postcard-start.html file, and follow the below instructions.

The HTML

The HTML is only slightly more involved than the example we used in [the first article of this guide](); it just has a few extra IDs and a title.

```
<form>
  <h1>to: Mozilla</h1>

  <div id="from">
    <label for="name">from:</label>
    <input type="text" id="name" name="user_name">
  </div>

  <div id="reply">
    <label for="mail">reply:</label>
    <input type="email" id="mail" name="user_email">
  </div>
```

```
  <div id="message">
    <label for="msg">Your message:</label>
    <textarea id="msg" name="user_message"></textarea>
  </div>

  <div class="button">
    <button type="submit">Send your message</button>
  </div>
</form>
```

Add the above code into the body of your HTML.

Organizing your assets

This is where the fun begins! Before we start coding, we need three additional assets:

1. The postcard background — download this image and save it in the same directory as your working HTML file.
2. A typewriter font: The "Secret Typewriter" font from fontsquirrel.com — download the TTF file into the same directory as above.
3. A handdrawn font: The "Journal" font from fontsquirrel.com — download the TTF file into the same directory as above.

Your fonts need some more processing before you start:

1. Go to the fontsquirrel Webfont Generator.
2. Using the form, upload both your font files and generate a webfont kit. Download the kit to your computer.
3. Unzip the provided zip file.
4. Inside the unzipped contents you will find two `.woff` files and two `.woff2` files. Copy these four files into a directory called fonts, in the same directory as before. We are using two different files for each font to maximise browser compatibility; see our Web fonts article for a lot more information.

The CSS

Now we can dig into the CSS for the example. Add all the code blocks shown below inside the `<style>` element, one after another.

First, we prepare the ground by defining our `@font-face` rules, all the basics on the `<body>` element, and the `<form>` element:

```
@font-face {
    font-family: 'handwriting';
    src: url('fonts/journal-webfont.woff2') format('woff2'),
         url('fonts/journal-webfont.woff') format('woff');
```

```
    font-weight: normal;
    font-style: normal;
}

@font-face {
    font-family: 'typewriter';
    src: url('fonts/veteran_typewriter-webfont.woff2') format('woff2'),
         url('fonts/veteran_typewriter-webfont.woff') format('woff');
    font-weight: normal;
    font-style: normal;
}

body {
  font  : 21px sans-serif;

  padding : 2em;
  margin  : 0;

  background : #222;
}

form {
  position: relative;

  width  : 740px;
  height : 498px;
  margin : 0 auto;

  background: #FFF url(background.jpg);
}
```

Now we can position our elements, including the title and all the form elements:

```
h1 {
  position : absolute;
  left : 415px;
  top  : 185px;

  font : 1em "typewriter", sans-serif;
}

#from {
  position: absolute;
  left : 398px;
  top  : 235px;
}

#reply {
  position: absolute;
  left : 390px;
  top  : 285px;
```

```
}

#message {
  position: absolute;
  left : 20px;
  top  : 70px;
}
```

That's where we start working on the form elements themselves. First, let's ensure that the `<label>`s are given the right font:

```
label {
  font : .8em "typewriter", sans-serif;
}
```

The text fields require some common rules. Simply put, we remove their `borders` and `backgrounds`, and redefine their `padding` and `margin`:

```
input, textarea {
  font    : .9em/1.5em "handwriting", sans-serif;

  border  : none;
  padding : 0 10px;
  margin  : 0;
  width   : 240px;

  background: none;
}
```

When one of these fields gains focus, we highlight them with a light grey, transparent, background. Note that it's important to add the `outline` property, in order to remove the default focus highlight added by some browsers:

```
input:focus, textarea:focus {
  background   : rgba(0,0,0,.1);
  border-radius: 5px;
  outline      : none;
}
```

Now that our text fields are complete, we need to adjust the display of the single and multiple line text fields to match, since they won't typically look the same using the defaults.

The single-line text field needs some tweaks to render nicely in Internet Explorer. Internet Explorer does not define the height of the fields based on the natural height of the font (which is the behavior of all other browsers). To fix this, we need to add an explicit height to the field, as follows:

```
input {
    height: 2.5em; /* for IE */
    vertical-align: middle; /* This is optional but it makes legacy IEs look better
*/
}
```

`<textarea>` elements default to being rendered as a block element. The two important things here are the `resize` and `overflow` properties. Because our design is a fixed-size design, we will use the `resize` property to prevent users from resizing our multi-line text field. The `overflow` property is used to make the field render more consistently across browsers. Some browsers default to the value `auto`, while some default to the value `scroll`. In our case, it's better to be sure every one will use `auto`:

```
textarea {
  display : block;

  padding : 10px;
  margin  : 10px 0 0 -10px;
  width   : 340px;
  height  : 360px;

  resize  : none;
  overflow: auto;
}
```

The `<button>` element is really convenient with CSS; you can do whatever you want, even using pseudo-elements:

```
button {
  position     : absolute;
  left         : 440px;
  top          : 360px;

  padding      : 5px;

  font         : bold .6em sans-serif;
  border       : 2px solid #333;
  border-radius: 5px;
  background   : none;

  cursor       : pointer;

-webkit-transform: rotate(-1.5deg);
   -moz-transform: rotate(-1.5deg);
    -ms-transform: rotate(-1.5deg);
     -o-transform: rotate(-1.5deg);
        transform: rotate(-1.5deg);
}

button:after {
```

```
  content: " >>>";
}

button:hover,
button:focus {
  outline    : none;
  background: #000;
  color    : #FFF;
}
```

And voila!

Note: If your example does not work quite like you expected and you want to check it against our version, you can find it on GitHub — see it running live (also see the source code).

Advanced styling for HTML forms

Previous Overview: FormsNext

In this article, we will see how to use CSS with HTML forms to style some form widgets that are difficult to customize. As we saw in the previous article, text fields and buttons are perfectly okay with CSS. Now we will dig into the dark part of HTML form styling.

Before going further, let's recall two kinds of HTML form widgets:

The bad

Elements that can hardly be styled, and require some complicated tricks, sometimes involving advanced CSS3 knowledge.

The ugly

Forget using CSS to style these elements. At best, you'll be able to do a few things, but it will not be reliable across browsers, and it will never be possible to take full control over their appearance.

CSS expressiveness

The main problem with form widgets, other than text fields and buttons, is that in many cases, CSS is not expressive enough to properly style complex widgets.

The recent evolution of HTML and CSS have extended CSS expressiveness:

- CSS 2.1 was very limited and gave us only three pseudo-classes:
  - `:active`
  - `:focus`
  - `:hover`
- CSS Selector Level 3 added a few new pseudo-classes, related to HTML forms:
  - `:enabled`
  - `:disabled`
  - `:checked`
  - `:indeterminate`
- CSS Basic UI Level 3 also adds several further pseudo-classes, to describe the state of a widget:
  - `:default`
  - `:valid`
  - `:invalid`
  - `:in-range`
  - `:out-of-range`
  - `:required`
  - `:optional`
  - `:read-only`
  - `:read-write`
- CSS Selector Level 4 which is currently under active development and heavy discussion, doesn't plan to add much to improve forms:
  - `:user-error` which is just an improvement of the `:invalid` pseudo-class.

All of this is a good start, but there are two issues with this. First, some browsers do not necessarily implement features beyond CSS 2.1. Second, these are simply not good enough for styling complex widgets, such as date pickers.

There are some experiments by browser vendors to extend CSS expressiveness about forms, and in some cases it's good to know what's available.

Warning: Even though these experiments are interesting, **they're not standard, which means it's not reliable**. If you use them (and you probably

often shouldn't), you do so at your own risk and you're doing something that may be bad for the Web by using non-standard properties.

- Mozilla CSS Extensions
  - `:-moz-placeholder`
  - `:-moz-submit-invalid`
  - `:-moz-ui-invalid`
  - `:-moz-ui-valid`
- WebKit CSS Extensions
  - `::-webkit-input-placeholder`
  - And many more
- Microsoft CSS Extensions
  - `:-ms-input-placeholder`

Controlling the appearance of form elements

WebKit- (Chrome, Safari) and Gecko- (Firefox) based browsers offer the highest degree of customization for HTML widgets. They are also available cross-platform, so they need a mechanism to switch from widgets with native look and feel, to widgets that are stylable by the user.

To that end, they use a proprietary property: `-webkit-appearance` or `-moz-appearance`. **Those properties are not standard and should not be used**. In fact, they even behave differently between WebKit and Gecko. However, there is one value that is good to know: `none`. With this value, you are able to gain (almost full) control over the style of a given widgets.
So, if you have trouble applying a style to an element, try using those proprietary properties. We'll see some examples below, but the best known use case for this property is for styling search fields on WebKit browsers:

```
<form>
    <input type="search">
</form>
<style>
input[type=search] {
    border: 1px dotted #999;
    border-radius: 0;

    -webkit-appearance: none;
}
</style>
```

Open in CodePenOpen in JSFiddle

Note: It's always hard to predict the future, when we talk about Web technologies. Extending CSS expressiveness is difficult, and there is some exploratory work with other specifications, such as Shadow DOM that offer some perspective. The quest for the fully stylable form is far from over.

Examples

Check boxes and radio buttons

Styling a check box or a radio button, by itself is kind of messy. For example, the sizes of check boxes and radio buttons are not really meant to be changed, and browsers can react very differently, if you try to do it.

A simple test case

Let's consider the following test case:

```
<span><input type="checkbox"></span>
span {
    display: inline-block;
    background: red;
}

input[type=checkbox] {
    width : 100px;
    height: 100px;
}
```

Here is the way different browsers handle this:

Browser	Rendering
Firefox 57 (Mac OSX)	
Firefox 57 (Windows 10)	

Browser	Rendering
Chrome 63 (Mac OSX)	
Chrome 63 (Windows 10)	
Opera 49 (Mac OSX)	
Internet Explorer 11 (Windows 10)	
Edge 16 (Windows 10)	

A more complex example

Because Opera and Internet Explorer do not have features such as `-webkit-appearance`or `-moz-appearance`, using them is not suitable. Fortunately, we are in a case where CSS is expressive enough to find solutions. Let's take a common example:

```
<form>
  <fieldset>
    <p>
      <input type="checkbox" id="first" name="fruit-1" value="cherry">
      <label for="first">I like cherry</label>
    </p>
    <p>
      <input type="checkbox" id="second" name="fruit-2" value="banana" disabled>
      <label for="second">I can't like banana</label>
    </p>
    <p>
      <input type="checkbox" id="third" name="fruit-3" value="strawberry">
      <label for="third">I like strawberry</label>
    </p>
  </fieldset>
</form>
```

with some basic styling:

```
body {
  font: 1em sans-serif;
}

form {
  display: inline-block;

  padding: 0;
  margin : 0;
}

fieldset {
  border : 1px solid #CCC;
  border-radius: 5px;
  margin : 0;
  padding: 1em;
}

label {
  cursor : pointer;
}

p {
  margin : 0;
}

p+p {
  margin : .5em 0 0;
```

```
}
```

Now, let's style this to have a custom check box.

The plan is to replace the native checkbox with an image of our own. First, we need to prepare an image with all the states required by a check box. Those states are: unchecked, checked, disabled unchecked, and disabled checked. This image will be used as a CSS sprite:

Let's start by hiding the original check boxes. We will simply move them outside the page viewport. There are two important things to consider here:

- Do not use `display:none` to hide the check box, because as we'll see below, we need the check box to be available to the user. With `display:none`, the check box is no longer accessible to the user, which means that it's impossible to check or uncheck it.
- We will use some CSS3 selectors to perform our styling. In order to support legacy browsers, we can prefix all our selectors with the `:root` pseudo-class. In the current state of implementation, all browsers that support what we need also support the `:root` pseudo-class, but others don't. This is an example of a convenient way to filter legacy Internet Explorer. Those browsers will see the regular check box while modern browsers will see the custom check box.

```
:root input[type=checkbox] {
  /* original check box are push outside the viewport */
  position: absolute;
  left: -1000em;
}
```

Now that we're rid of the native check box, let's add our own. To that end, we will use the `:before` pseudo element of the `<label>` element that follows the original check box. So in the following selector, we use the attribute selector to target the check box, then we use the adjacent sibling selector to target the `label` following the original check box. Finally, we access the `:before` pseudo-element and style it to have it display our custom unchecked check box.

```
:root input[type=checkbox] + label:before {
  content: "";
```

```
  display: inline-block;
  width  : 16px;
  height : 16px;
  margin : 0 .5em 0 0;
  background: url("https://developer.mozilla.org/files/4173/checkbox-sprite.png") no-repeat 0 0;

/* The following is used to adjust the position of
   the check boxes on the text baseline */

  vertical-align: bottom;
  position: relative;
  bottom: 2px;
}
```

We use the `:checked` and `:disabled` pseudo-classes on the original check box to change the state of our custom check box accordingly. Because we're using a CSS sprite, all we need to do is change the position of the background.

```
:root input[type=checkbox]:checked + label:before {
  background-position: 0 -16px;
}

:root input[type=checkbox]:disabled + label:before {
  background-position: 0 -32px;
}

:root input[type=checkbox]:checked:disabled + label:before {
  background-position: 0 -48px;
}
```

The last (but very important) thing: when a user uses the keyboard to navigate from one form widget to another, each widget should be focused visually. Because we hide the native check boxes, we have to implement this feature ourselves, to let the user know where they are in the form. The following CSS implements the focusing of our custom checkboxes.

```
:root input[type=checkbox]:focus + label:before {
  outline: 1px dotted black;
}
```

You can see the live result:

Open in CodePenOpen in JSFiddle

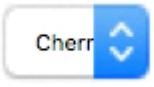
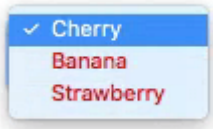
Dealing with the select nightmare

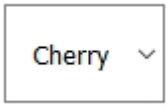
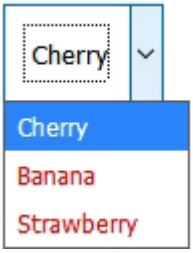
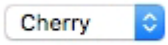
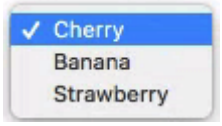
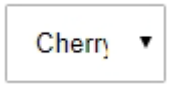
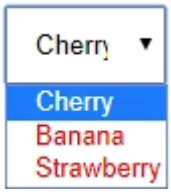
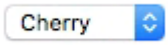
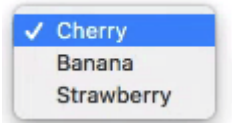
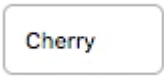
The `<select>` element is considered an "ugly" widget, because it's impossible to style it consistently cross platform. However, some things are possible. Rather than a long explanation, let's look at an example:

```
<select>
  <option>Cherry</option>
  <option>Banana</option>
  <option>Strawberry</option>
</select>
select {
  width   : 80px;
  padding : 10px;
}

option {
  padding : 5px;
  color   : red;
}
```

The following table shows how different browsers handle this, in two cases. The first two columns are just the above example. The second two columns use additional custom CSS, to gain more control on the widget's appearance:

```
select, option {
  -webkit-appearance : none; /* To gain control over the appearance on
WebKit/Chromium */
  -moz-appearance : none; /* To gain control over the appearance on Gecko */

  /* To gain control over the appearance on and Trident (IE)
     Note that it also works on Gecko and has partial effects on WebKit */
  background : none;
}
```

Browser	Regular rendering			
	closed	open	closed	
Firefox 57 (Mac OSX)				

Browser	Regular rendering		
	closed	open	closed
Firefox 57 (Windows 10)			
Chrome 63 (Mac OSX)			
Chrome 63 (Windows 10)			
Opera 49 (Mac OSX)			
IE11 (Windows 10)			
Edge 16 (Windows 10)			

As you can see, even with the help of the `-*-appearance` properties, there are still some remaining issues:

- The `padding` property is handled inconsistently across operating systems and browsers.
- Legacy Internet Explorer does not allow smooth styling.
- Firefox does not have a way to style the dropdown arrow.
- If you want to style the `<option>` elements inside the dropdown list, the behavior of Chrome and Opera vary from one system to another.
  Also, with our example, we are only using three CSS properties. Imagine the mess when even more CSS properties are considered. As you can see, CSS is not suitable for changing the look and feel of these widgets consistently, but it still lets you tweak some things. As long as you're willing to live with differences, from one browser and one operating system to another.

  We will help understand which properties are suitable in the next article: Properties compatibility table for form widgets.

The road to nicer forms: useful libraries and polyfills

Although CSS is expressive enough for check boxes and radio button, it is far from true for more advanced widgets. Even though a few things are possible with the `<select>`element, the file widget cannot be styled at all. The same goes for the date picker, etc.
If you want to gain full control over form widgets, you have no choice but to rely on JavaScript. In the article How to build custom form widgets we will see how to do it on our own, but there are some very useful libraries out there that can help you:

- Uni-form is a framework that standardizes form markup, styling it with CSS. It also offers a few additional features when used with jQuery, but that's optional.
- Formalize is an extension to common JavaScript frameworks (such as jQuery, Dojo, YUI, etc.) that helps to normalize and customize your forms.
- Niceforms is a standalone JavaScript method that provides complete customization of web forms. You can use some of the built in themes, or create your own.
  The following libraries aren't just about forms, they have very interesting features for dealing with HTML forms:

- jQuery UI offers some very interesting advanced and customizable widgets, such as date pickers (with special attention given to accessibility).
- Twitter Bootstrap can be really helpful if you want to normalize your forms.
- WebShim is a huge tool that can help you deal with browser HTML5 support. The web forms part can be really helpful.
  Remember that binding CSS and JavaScript can have side effects. So if you choose to use one of those libraries, you should always have fallback style sheets in case the script fails. There are many reasons why scripts may fail, especially in the mobile world, and you need to design your Web site or app to handle these cases as best as possible.

The following compatibility tables try to summarize the state of CSS support for HTML forms. Due to the complexity of CSS and HTML forms, these tables can't be considered a perfect reference. However, they will give you good insight into what can and can't be done, which will help you learn how to do things.

How to read the tables

Values

For each property, there are four possible values:

YES
: There's reasonably consistent support for the property across browsers. You may still face strange side effects in certain edge cases.

PARTIAL
: While the property works, you may frequently face strange side effects or inconsistencies. You should probably avoid these properties unless you master those side effects first.

NO
: The property simply doesn't work or is so inconsistent that it's not reliable.

N.A.
: The property has no meaning for this type of widget.

Rendering

For each property there are two possible renderings:

N (Normal)
: Indicates that the property is applied as it is

T (Tweaked)
: Indicates that the property is applied with the extra rule below:

```
* {
/* This turn off the native look and feel on WebKit based browsers */
  -webkit-appearance: none;

/* This turn off the native look and feel on Gecko based browsers */
  -moz-appearance: none;

/* This turn off the native look and feel on several different browsers
   including Opera, Internet Explorer and Firefox */
  background: none;
}
```

Compatibility tables

Global behaviors

Some behaviors are common to many browsers at a global level:

`border`, `background`, `border-radius`, `height`
: Using one of these properties can partially or fully turn off the native look & feel of widgets on some browsers. Be careful when you use them.

`line-height`
: This property is supported inconsistently across browsers and you should avoid it.

`text-decoration`
: This property is not supported by Opera on form widgets.

`text-overflow`
: Opera, Safari, and IE9 do not support this property on form widgets.

`text-shadow`
: Opera does not support `text-shadow` on form widgets and IE9 does not support it at all.

Text fields

Property	N	T	Note
CSS box model			
`width`	Yes	Yes	
`height`	Partial[1][2]	Yes	WebKit browsers (mostly on Mac OSX and iOS) use search fields. Therefore, it's required to use `-webkit-a` apply this property to search fields. On Windows 7, Internet Explorer 9 does not apply th unless `background:none` is applied.
`border`	Partial[1][2]	Yes	WebKit browsers (mostly on Mac OSX and iOS) use search fields. Therefore, it's required to use `-webkit-a` apply this property to search fields. On Windows 7, Internet Explorer 9 does not apply th unless `background:none` is applied.
`margin`	Yes	Yes	
`padding`	Partial[1][2]	Yes	WebKit browsers (mostly on Mac OSX and iOS) use search fields. Therefore, it's required to use `-webkit-a` apply this property to search fields. On Windows 7, Internet Explorer 9 does not apply th unless `background:none` is applied.
Text and font			
`color`[1]	Yes	Yes	If the `border-color` property is not set, some WebKit b the `color` property to the border as well as the font or
`font`	Yes	Yes	See the note about `line-height`
`letter-spacing`	Yes	Yes	
`text-align`	Yes	Yes	
`text-decoration`	Partial	Partial	See the note about Opera

Property	N	T	Note
text-indent	Partial[1]	Partial[1]	IE9 supports this property only on <textarea>s, where single line text fields.
text-overflow	Partial	Partial	
text-shadow	Partial	Partial	
text-transform	Yes	Yes	
Border and background			
background	Partial[1]	Yes	WebKit browsers (mostly on Mac OSX and iOS) use search fields. Therefore, it's required to use -webkit- apply this property to search fields. On Windows 7, apply the border unless background:none is applied.
border-radius	Partial[1][2]	Yes	WebKit browsers (mostly on Mac OSX and iOS) use search fields. Therefore, it's required to use -webkit- apply this property to search fields. On Windows 7, apply the border unless background:none is applied. On Opera the border-radius property is applied only if
box-shadow	No	Partial[1]	IE9 does not support this property.

Buttons

Property	N	T	Note
CSS box model			
width	Yes	Yes	
height	Partial[1]	Yes	This property is not applied on WebKit based brows
border	Partial	Yes	
margin	Yes	Yes	
padding	Partial[1]	Yes	This property is not applied on WebKit based brows
Text and font			
color	Yes	Yes	
font	Yes	Yes	See the note about line-height.

Property	N	T	Note
`letter-spacing`	Yes	Yes	
`text-align`	No	No	
`text-decoration`	Partial	Yes	
`text-indent`	Yes	Yes	
`text-overflow`	No	No	
`text-shadow`	Partial	Partial	
`text-transform`	Yes	Yes	
Border and background			
`background`	Yes	Yes	
`border-radius`	Yes[1]	Yes[1]	On Opera the `border-radius` property is applied only if
`box-shadow`	No	Partial[1]	IE9 does not support this property.

Number

On browsers that implement the `number` widget, there is no standard way to change the style of spinners used to change the value of the field. It is worth noting that the spinners on Safari are outside the field.

Property	N	T	Note
CSS box model			
`width`	Yes	Yes	
`height`	Partial[1]	Partial[1]	On Opera, the spinners are zoomed in, which can hi
`border`	Yes	Yes	
`margin`	Yes	Yes	
`padding`	Partial[1]	Partial[1]	On Opera, the spinners are zoomed in, which can hi
Text and font			
`color`	Yes	Yes	
`font`	Yes	Yes	See the note about `line-height`.

Property	N	T	Note
letter-spacing	Yes	Yes	
text-align	Yes	Yes	
text-decoration	Partial	Partial	
text-indent	Yes	Yes	
text-overflow	No	No	
text-shadow	Partial	Partial	
text-transform	N.A.	N.A.	
Border and background			
background	No	No	Supported but there is too much inconsistency betwe
border-radius	No	No	
box-shadow	No	No	

Check boxes and radio buttons

Property	N	T	Note
CSS box model			
width	No[1]	No[1]	Some browsers add extra margins and others stretch
height	No[1]	No[1]	Some browsers add extra margins and others stretch
border	No	No	
margin	Yes	Yes	
padding	No	No	
Text and font			
color	N.A.	N.A.	
font	N.A.	N.A.	
letter-spacing	N.A.	N.A.	
text-align	N.A.	N.A.	

Property	N	T	Note
text-decoration	N.A.	N.A.	
text-indent	N.A.	N.A.	
text-overflow	N.A.	N.A.	
text-shadow	N.A.	N.A.	
text-transform	N.A.	N.A.	
Border and background			
background	No	No	
border-radius	No	No	
box-shadow	No	No	

Select boxes (single line)

Firefox does not provide any way to change the down arrow on the `<select>` element.

Property	N	T	Note
CSS box model			
width	Partial[1]	Partial[1]	This property is okay on the `<select>` element, but i the `<option>` or `<optgroup>`elements.
height	No	Yes	
border	Partial	Yes	
margin	Yes	Yes	
padding	No[1]	Partial[2]	The property is applied, but in an inconsistent way OSX. The property is well applied on the `<select>` elemen handled on `<option>` and `<optgroup>` elements.
Text and font			
color	Partial[1]	Partial[1]	On Mac OSX, WebKit based browsers do not supp widgets and they, along with Opera, do not suppor on `<option>` and `<optgroup>` elements.

Property	N	T	Note
font	Partial[1]	Partial[1]	On Mac OSX, WebKit based browsers do not supp widgets and they, along with Opera, do not suppor on <option> and <optgroup> elements.
letter-spacing	Partial[1]	Partial[1]	IE9 does not support this property on <select>, <opt WebKit based browsers on Mac OSX do not suppo on <option> and <optgroup> elements.
text-align	No[1]	No[1]	IE9 on Windows 7 and WebKit based browsers on property on this widget.
text-decoration	Partial[1]	Partial[1]	Only Firefox provides full support for this property. property at all and other browsers only support it o
text-indent	Partial[1][2]	Partial[1][2]	Most of the browsers only support this property on IE9 does not support this property.
text-overflow	No	No	
text-shadow	Partial[1][2]	Partial[1][2]	Most of the browsers only support this property on IE9 does not support this property.
text-transform	Partial[1]	Partial[1]	Most of the browsers only support this property on
Border and background			
background	Partial[1]	Partial[1]	Most of the browsers only support this property on
border-radius	Partial[1]	Partial[1]	
box-shadow	No	Partial[1]	

Select boxes (multiline)

Property	N	T	Note
CSS box model			
width	Yes	Yes	
height	Yes	Yes	
border	Yes	Yes	
margin	Yes	Yes	

Property	N	T	Note
padding	Partial[1]	Partial[1]	Opera does not support padding-top and padding-bottom
Text and font			
color	Yes	Yes	
font	Yes	Yes	See the note about line-height.
letter-spacing	Partial[1]	Partial[1]	IE9 does not support this property on <select>, <option> WebKit based browsers on Mac OSX do not support t on <option> and <optgroup> elements.
text-align	No[1]	No[1]	IE9 on Windows 7 and WebKit based browser on Mac property on this widget.
text-decoration	No[1]	No[1]	Only supported by Firefox and IE9+.
text-indent	No	No	
text-overflow	No	No	
text-shadow	No	No	
text-transform	Partial[1]	Partial[1]	Most of the browsers only support this property on the
Border and background			
background	Yes	Yes	
border-radius	Yes[1]	Yes[1]	On Opera the border-radius property is applied only if a
box-shadow	No	Partial[1]	IE9 does not support this property.

Datalist

Property	N	T
CSS box model		
width	No	No
height	No	No
border	No	No
margin	No	No

Property	N	T
padding	No	No
Text and font		
color	No	No
font	No	No
letter-spacing	No	No
text-align	No	No
text-decoration	No	No
text-indent	No	No
text-overflow	No	No
text-shadow	No	No
text-transform	No	No
Border and background		
background	No	No
border-radius	No	No
box-shadow	No	No

File picker

Property	N	T	Note
CSS box model			
width	No	No	
height	No	No	
border	No	No	
margin	Yes	Yes	
padding	No	No	
Text and font			
color	Yes	Yes	

Property	N	T	Note
font	No[1]	No[1]	Supported, but there is too much inconsistency betwe
letter-spacing	Partial[1]	Partial[1]	Many browsers apply this property to the select butto
text-align	No	No	
text-decoration	No	No	
text-indent	Partial[1]	Partial[1]	It acts more or less like an extra left margin outside th
text-overflow	No	No	
text-shadow	No	No	
text-transform	No	No	
Border and background			
background	No[1]	No[1]	Supported, but there is too much inconsistency betwe
border-radius	No	No	
box-shadow	No	Partial[1]	IE9 does not support this property.

Date pickers

Many properies are supported but there is to much inconstency between browsers to be reliable.

Property	N	T
CSS box model		
width	No	No
height	No	No
border	No	No
margin	Yes	Yes
padding	No	No
Text and font		
color	No	No

Property	N	T
font	No	No
letter-spacing	No	No
text-align	No	No
text-decoration	No	No
text-indent	No	No
text-overflow	No	No
text-shadow	No	No
text-transform	No	No
Border and background		
background	No	No
border-radius	No	No
box-shadow	No	No

Color pickers

There is currently not enough implementation to get realiable behaviors.

Property	N	T	Note
CSS box model			
width	Yes	Yes	
height	No[1]	Yes	Opera handles this like a select widget with the same restr
border	Yes	Yes	
margin	Yes	Yes	
padding	No[1]	Yes	Opera handles this like a select widget with the same restr
Text and font			
color	N.A.	N.A.	
font	N.A.	N.A.	

Property	N	T	Note
letter-spacing	N.A.	N.A.	
text-align	N.A.	N.A.	
text-decoration	N.A.	N.A.	
text-indent	N.A.	N.A.	
text-overflow	N.A.	N.A.	
text-shadow	N.A.	N.A.	
text-transform	N.A.	N.A.	
Border and background			
background	No[1]	No[1]	
border-radius	No[1]	No[1]	Supported, but there is too much inconsistency between b
box-shadow	No[1]	No[1]	

Meters and progress

There is currently not enough implementation to get realiable behaviors.

Property	N	T	Note
CSS box model			
width	Yes	Yes	
height	Yes	Yes	
border	Partial	Yes	
margin	Yes	Yes	
padding	Yes	Partial[1]	Chrome hides the `<progress>` and `<meter>` element when t on a tweaked element.
Text and font			
color	N.A.	N.A.	
font	N.A.	N.A.	
letter-spacing	N.A.	N.A.	

Property	N	T	Note
`text-align`	N.A.	N.A.	
`text-decoration`	N.A.	N.A.	
`text-indent`	N.A.	N.A.	
`text-overflow`	N.A.	N.A.	
`text-shadow`	N.A.	N.A.	
`text-transform`	N.A.	N.A.	
Border and background			
`background`	No[1]	No[1]	Supported, but there is too much inconsistency betwee
`border-radius`	No[1]	No[1]	
`box-shadow`	No[1]	No[1]	

Range

There is no standard way to change the style of the range grip and Opera has no way to tweak the default rendering of the range widget.

Property	N	T	Note
CSS box model			
`width`	Yes	Yes	
`height`	Partial[1]	Partial[1]	Chrome and Opera add some extra space around the Windows 7 stretches the range grip.
`border`	No	Yes	
`margin`	Yes	Yes	
`padding`	Partial[1]	Yes	The `padding` is applied, but has no visual effect.
Text and font			
`color`	N.A.	N.A.	
`font`	N.A.	N.A.	
`letter-spacing`	N.A.	N.A.	

Property	N	T	Note
text-align	N.A.	N.A.	
text-decoration	N.A.	N.A.	
text-indent	N.A.	N.A.	
text-overflow	N.A.	N.A.	
text-shadow	N.A.	N.A.	
text-transform	N.A.	N.A.	
Border and background			
background	No[1]	No[1]	Supported, but there is too much inconsistency betwe
border-radius	No[1]	No[1]	
box-shadow	No[1]	No[1]	

Image buttons

Property	N	T	Note
CSS box model			
width	Yes	Yes	
height	Yes	Yes	
border	Yes	Yes	
margin	Yes	Yes	
padding	Yes	Yes	
Text and font			
color	N.A.	N.A.	
font	N.A.	N.A.	
letter-spacing	N.A.	N.A.	
text-align	N.A.	N.A.	
text-decoration	N.A.	N.A.	
text-indent	N.A.	N.A.	

Property	N	T	Note
text-overflow	N.A.	N.A.	
text-shadow	N.A.	N.A.	
text-transform	N.A.	N.A.	
Border and background			
background	Yes	Yes	
border-radius	Partial[1]	Partial[1]	IE9 does not support this prop
box-shadow	Partial[1]	Partial[1]	IE9 does not support this prop