

Getting started with HTML

In this article, we cover the absolute basics of HTML. To get you started, this article defines elements, attributes, and all the other important terms you may have heard. It also explains where these fit into HTML. You will learn how HTML elements are structured, how a typical HTML page is structured, and other important basic language features. Along the way, there will be an opportunity to play with HTML too!

Prerequisites:	Basic software installed , and basic knowledge of working with files .
Objective:	To gain basic familiarity with HTML, and practice writing a few HTML elements.

What is HTML?

[HTML](#) (HyperText Markup Language) is a *markup language* that tells web browsers how to structure the web pages you visit. It can be as complicated or as simple as the web developer wants it to be. HTML consists of a series of [elements](#), which you use to enclose, wrap, or *mark up* different parts of content to make it appear or act in a certain way. The enclosing [tags](#) can make content into a hyperlink to connect to another page, italicize words, and so on. For example, consider the following line of text:

My cat is very grumpy

If we wanted the text to stand by itself, we could specify that it is a paragraph by enclosing it in a paragraph (`<p>`) element:

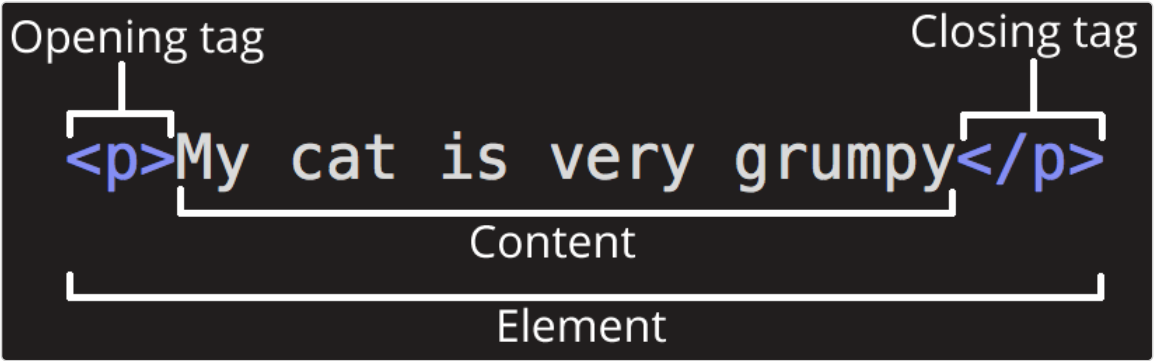
HTML

```
<p>My cat is very grumpy</p>
```

Note: Tags in HTML are not case-sensitive. This means they can be written in uppercase or lowercase. For example, a `<title>` tag could be written as `<title>`, `<TITLE>`, `<Title>`, `<TiTlE>`, etc., and it will work. However, it is best practice to write all tags in lowercase for consistency and readability.

Anatomy of an HTML element

Let's further explore our paragraph element from the previous section:



The anatomy of our element is:

- **The opening tag:** This consists of the name of the element (in this example, *p* for paragraph), wrapped in opening and closing angle brackets. This opening tag marks where the element begins or starts to take effect. In this example, it precedes the start of the paragraph text.
- **The content:** This is the content of the element. In this example, it is the paragraph text.
- **The closing tag:** This is the same as the opening tag, except that it includes a forward slash before the element name. This marks where the element ends. Failing to include a closing tag is a common beginner error that can produce peculiar results.

The element is the opening tag, followed by content, followed by the closing tag.

Active learning: creating your first HTML element

Edit the line below in the "Editable code" area by wrapping it with the tags `` and ``. To *open the element*, put the opening tag `` at the start of the line. To *close the element*, put the closing tag `` at the end of the line. Doing this should give the line italic text formatting! See your changes update live in the *Output* area.

If you make a mistake, you can clear your work using the *Reset* button. If you get really stuck, press the *Show solution* button to see the answer.

Play

Nesting elements

Elements can be placed within other elements. This is called *nesting*. If we wanted to state that our cat is **very** grumpy, we could wrap the word *very* in a `` element, which means that the word is to have strong(er) text formatting:

HTML

```
<p>My cat is <strong>very</strong> grumpy.</p>
```

There is a right and wrong way to do nesting. In the example above, we opened the `p` element first, then opened the `strong` element. For proper nesting, we should close the `strong` element first, before closing the `p`.

The following is an example of the *wrong* way to do nesting:

HTML

```
<p>My cat is <strong>very grumpy.</p></strong>
```

The **tags have to open and close in a way that they are inside or outside one another**. With the kind of overlap in the example above, the browser has to guess at your intent. This kind of guessing can result in unexpected results.

Void elements

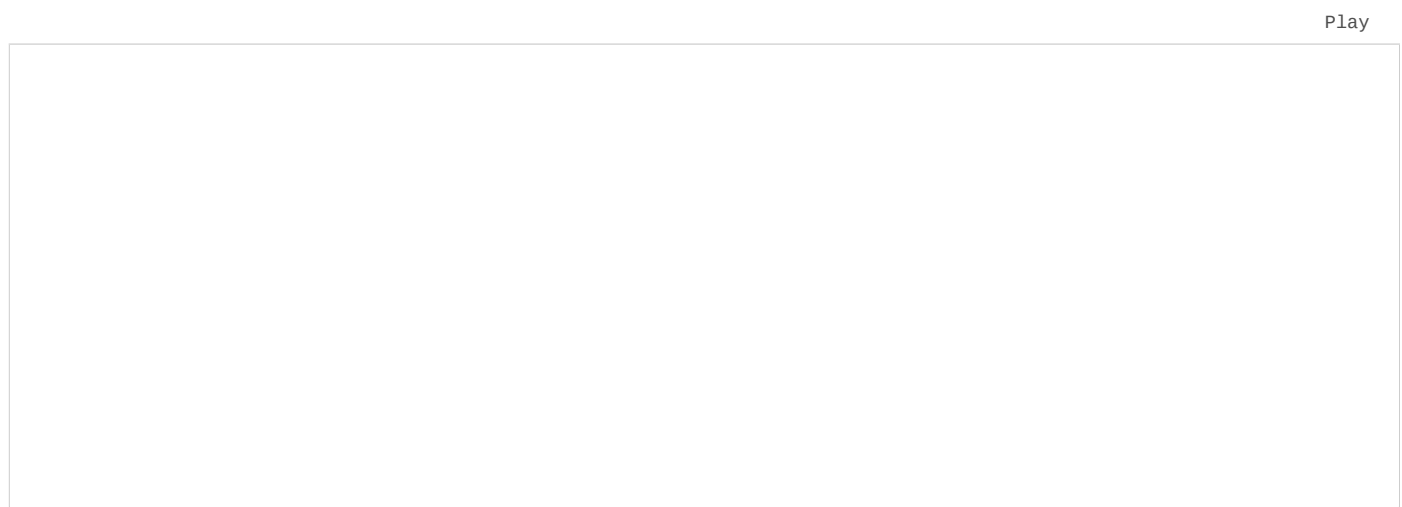
Not all elements follow the pattern of an opening tag, content, and a closing tag. Some elements consist of a single tag, which is typically used to insert/embed something in the document. Such elements are called [void elements](#). For example, the `` element embeds an image file onto a page:

HTML Play

```

```

This would output the following:



Note: In HTML, there is no requirement to add a `/` at the end of a void element's tag, for example: ``. However, it is also a valid syntax, and you may do this when you want your HTML to be valid XML.

Attributes

Elements can also have attributes. Attributes look like this:

```
<p class="editor-note">My cat is very grumpy</p>
```

Attributes contain extra information about the element that won't appear in the content. In this example, the `class` attribute is an identifying name used to target the element with style information.

An attribute should have:

- A space between it and the element name. (For an element with more than one attribute, the attributes should be separated by spaces too.)

- The attribute name, followed by an equal sign.
- An attribute value, wrapped with opening and closing quote marks.

Active learning: Adding attributes to an element

The `` element can take a number of attributes, including:

`src`

The `src` attribute is a **required** attribute that specifies the location of the image. For example:

```
src="https://raw.githubusercontent.com/mdn/beginner-html-site/gh-pages/images/firefox-icon.png" .
```

`alt`

The `alt` attribute specifies a text description of the image. For example: `alt="The Firefox icon"` .

`width`

The `width` attribute specifies the width of the image with the unit being pixels. For example: `width="300"` .

`height`

The `height` attribute specifies the height of the image with the unit being pixels. For example: `height="300"` .

Edit the line below in the *Input* area to turn it into an image.

1. Find your favorite image online, right click it, and press *Copy Image Link/Address*.
2. Back in the area below, add the `src` attribute and fill it with the link from step 1.
3. Set the `alt` attribute.
4. Add the `width` and `height` attributes.

You will be able to see your changes live in the *Output* area.

If you make a mistake, you can always reset it using the *Reset* button. If you get really stuck, press the *Show solution* button to see the answer.

Play

Boolean attributes

Sometimes you will see attributes written without values. This is entirely acceptable. These are called Boolean attributes. Boolean attributes can only have one value, which is generally the same as the attribute name. For example, consider the [disabled](#) attribute, which you can assign to form input elements. (You use this to *disable* the form input elements so the user can't make entries. The disabled elements typically have a grayed-out appearance.) For example:

HTML

Play

```
<input type="text" disabled="disabled" />
```

As shorthand, it is acceptable to write this as follows:

HTML

Play

```
<!-- using the disabled attribute prevents the end user from entering text into the input box -->
<input type="text" disabled />

<!-- text input is allowed, as it doesn't contain the disabled attribute -->
<input type="text" />
```

For reference, the example above also includes a non-disabled form input element. The HTML from the example above produces this result:

Play

Omitting quotes around attribute values

If you look at code for a lot of other sites, you might come across a number of strange markup styles, including attribute values without quotes. This is permitted in certain circumstances, but it can also break your markup in other circumstances. The element in the code snippet below, `<a>`, is called an anchor. Anchors enclose text and turn them into links. The `href` attribute specifies the web address the link points to. You can write this basic version below with *only* the `href` attribute, like this:

HTML

Play

```
<a href=https://www.mozilla.org/>favorite website</a>
```

Anchors can also have a `title` attribute, a description of the linked page. However, as soon as we add the `title` in the same fashion as the `href` attribute there are problems:

HTML

Play

```
<a href=https://www.mozilla.org/ title=The Mozilla homepage>favorite website</a>
```

As written above, the browser misinterprets the markup, mistaking the `title` attribute for three attributes: a title attribute with the value `The`, and two Boolean attributes, `Mozilla` and `homepage`. Obviously, this is not intended! It will cause errors or unexpected behavior, as you can see in the live example below. Try hovering over the link to view the title text!

Play

Always include the attribute quotes. It avoids such problems, and results in more readable code.

Single or double quotes?

In this article, you will also notice that the attributes are wrapped in double quotes. However, you might see single quotes in some HTML code. This is a matter of style. You can feel free to choose which one you prefer. Both of these lines are equivalent:

HTML

```
<a href='https://www.example.com'>A link to my example.</a>
```

```
<a href="https://www.example.com">A link to my example.</a>
```

Make sure you don't mix single quotes and double quotes. This example (below) shows a kind of mixing of quotes that will go wrong:

HTML

```
<a href="https://www.example.com">A link to my example.</a>
```

However, if you use one type of quote, you can include the other type of quote *inside* your attribute values:

HTML

```
<a href="https://www.example.com" title="Isn't this fun?">
  A link to my example.
</a>
```

To use quote marks inside other quote marks of the same type (single quote or double quote), use [HTML entities](#). For example, this will break:

HTML

```
<a href="https://www.example.com" title="An "interesting" reference">A link to my example.</a>
```

Instead, you need to do this:

HTML

```
<a href="https://www.example.com" title="An &quot;interesting&quot; reference">A link to my example.</a>
```

Anatomy of an HTML document

Individual HTML elements aren't very useful on their own. Next, let's examine how individual elements combine to form an entire HTML page:

HTML

```
<!doctype html>
<html lang="en-US">
  <head>
```

```
<meta charset="utf-8" />
<title>My test page</title>
</head>
<body>
  <p>This is my page</p>
</body>
</html>
```

Here we have:

1. `<!DOCTYPE html>`: The doctype. When HTML was young (1991-1992), doctypes were meant to act as links to a set of rules that the HTML page had to follow to be considered good HTML. Doctypes used to look something like this:

HTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

More recently, the doctype is a historical artifact that needs to be included for everything else to work right. `<!DOCTYPE html>` is the shortest string of characters that counts as a valid doctype. That is all you need to know!

2. `<html></html>`: The [<html>](#) element. This element wraps all the content on the page. It is sometimes known as the root element.
3. `<head></head>`: The [<head>](#) element. This element acts as a container for everything you want to include on the HTML page, **that isn't the content** the page will show to viewers. This includes keywords and a page description that would appear in search results, CSS to style content, character set declarations, and more. You will learn more about this in the next article of the series.
4. `<meta charset="utf-8">`: The [<meta>](#) element. This element represents metadata that cannot be represented by other HTML meta-related elements, like [<base>](#), [<link>](#), [<script>](#), [<style>](#) or [<title>](#). The [charset](#) attribute specifies the character encoding for your document as UTF-8, which includes most characters from the vast majority of human written languages. With this setting, the page can now handle any textual content it might contain. There is no reason not to set this, and it can help avoid some problems later.
5. `<title></title>`: The [<title>](#) element. This sets the title of the page, which is the title that appears in the browser tab the page is loaded in. The page title is also used to describe the page when it is bookmarked.
6. `<body></body>`: The [<body>](#) element. This contains *all* the content that displays on the page, including text, images, videos, games, playable audio tracks, or whatever else.

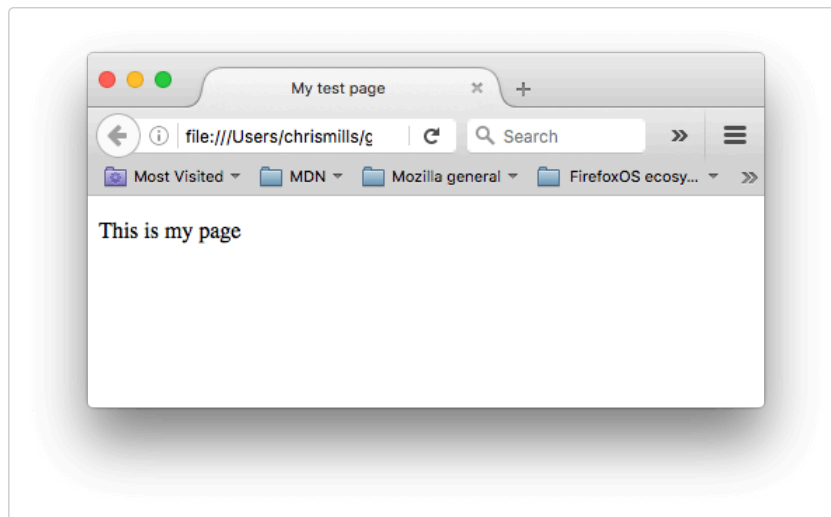
Active learning: Adding some features to an HTML document

If you want to experiment with writing some HTML on your local computer, you can:

1. Copy the HTML page example listed above.
2. Create a new file in your text editor.
3. Paste the code into the new text file.
4. Save the file as `index.html`.

Note: You can also find this basic HTML template on the [MDN Learning Area GitHub repo](#).

You can now open this file in a web browser to see what the rendered code looks like. Edit the code and refresh the browser to see what the result is. Initially, the page looks like this:

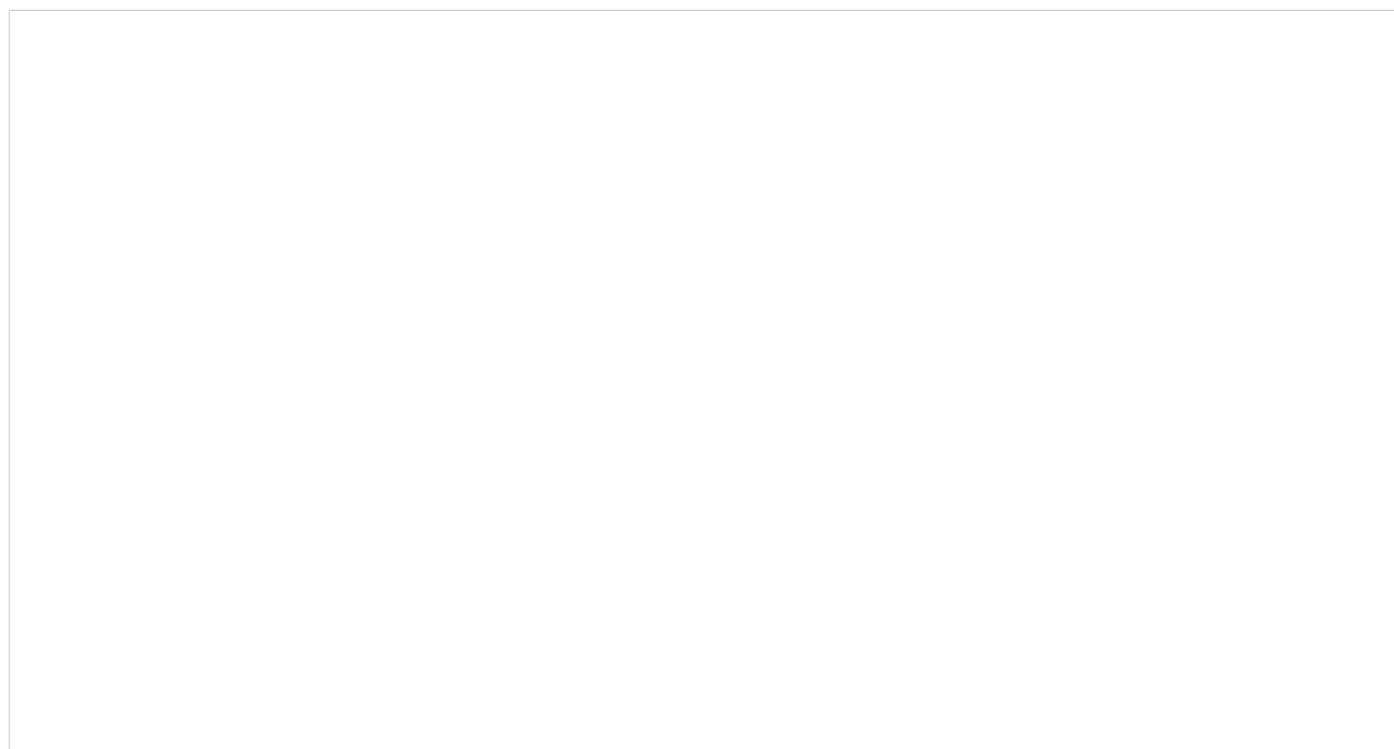


In this exercise, you can edit the code locally on your computer, as described previously, or you can edit it in the sample window below (the editable sample window represents just the contents of the `<body>` element, in this case). Sharpen your skills by implementing the following tasks:

- Just below the opening tag of the `<body>` element, add a main title for the document. This should be wrapped inside an `<h1>` opening tag and `</h1>` closing tag.
- Edit the paragraph content to include text about a topic that you find interesting.
- Make important words stand out in bold by wrapping them inside a `` opening tag and `` closing tag.
- Add a link to your paragraph, as [explained earlier in the article](#).
- Add an image to your document. Place it below the paragraph, as [explained earlier in the article](#). Earn bonus points if you manage to link to a different image (either locally on your computer or somewhere else on the web).

If you make a mistake, you can always reset it using the *Reset* button. If you get really stuck, press the *Show solution* button to see the answer.

Play



Whitespace in HTML

In the examples above, you may have noticed that a lot of whitespace is included in the code. This is optional. These two code snippets are equivalent:

HTMLPlay

```
<p id="nowhitespace">Dogs are silly.</p>

<p id="whitespace">Dogs
  are
    silly.</p>
```

No matter how much whitespace you use inside HTML element content (which can include one or more space characters, but also line breaks), the HTML parser reduces each sequence of whitespace to a single space when rendering the code. So why use so much whitespace? The answer is readability.

It can be easier to understand what is going on in your code if you have it nicely formatted. In our HTML we've got each nested element indented by two spaces more than the one it is sitting inside. It is up to you to choose the style of formatting (how many spaces for each level of indentation, for example), but you should consider formatting it.

Let's have a look at how the browser renders the two paragraphs above with and without whitespace:

Play

Note: Accessing the `innerHTML` of elements from JavaScript will keep all the whitespace intact. This may return unexpected results if the whitespace is trimmed by the browser.

JSPlay

```
const nowhitespace = document.getElementById("nowhitespace").innerHTML;
console.log(nowhitespace);
// "Dogs are silly."

const whitespace = document.getElementById("whitespace").innerHTML;
console.log(whitespace);
// "Dogs
//   are
//     silly."
```

Entity references: Including special characters in HTML

In HTML, the characters `<`, `>`, `"`, `'`, and `&` are special characters. They are parts of the HTML syntax itself. So how do you include one of these special characters in your text? For example, if you want to use an ampersand or less-than sign, and not have it interpreted as code.

You do this with character references. These are special codes that represent characters, to be used in these exact circumstances. Each character reference starts with an ampersand (`&`), and ends with a semicolon (`;`).

Literall character	Character reference equivalent
<	<

Literal character	Character reference equivalent
>	>
"	"
'	'
&	&

The character reference equivalent could be easily remembered because the text it uses can be seen as less than for `<`, quotation for `"`; and similarly for others. To find more about entity references, see [List of XML and HTML character entity references](#) (Wikipedia).

In the example below, there are two paragraphs:

HTMLPlay

```
<p>In HTML, you define a paragraph using the <p> element.</p>

<p>In HTML, you define a paragraph using the &lt;p&gt; element.</p>
```

In the live output below, you can see that the first paragraph has gone wrong. The browser interprets the second instance of `<p>` as starting a new paragraph. The second paragraph looks fine because it has angle brackets with character references.

Play

Note: You don't need to use entity references for any other symbols, as modern browsers will handle the actual symbols just fine as long as your HTML's [character encoding is set to UTF-8](#).

HTML comments

HTML has a mechanism to write comments in the code. Browsers ignore comments, effectively making comments invisible to the user. The purpose of comments is to allow you to include notes in the code to explain your logic or coding. This is very useful if you return to a code base after being away for long enough that you don't completely remember it. Likewise, comments are invaluable as different people are making changes and updates.

To write an HTML comment, wrap it in the special markers `<!--` and `-->`. For example:

HTMLPlay

```
<p>I'm not inside a comment</p>

<!-- <p>I am!</p> -->
```

As you can see below, only the first paragraph is displayed in the live output.

Summary

You made it to the end of the article! We hope you enjoyed your tour of the basics of HTML.

At this point, you should understand what HTML looks like, and how it works at a basic level. You should also be able to write a few elements and attributes. The subsequent articles of this module go further on some of the topics introduced here, as well as presenting other concepts of the language.

- As you start to learn more about HTML, consider learning the basics of CSS (Cascading Style Sheets). [CSS](#) is the language used to style web pages, such as changing fonts or colors or altering the page layout. HTML and CSS work well together, as you will soon discover.

See also

- [Applying color to HTML elements using CSS](#)

Help improve MDN

Was this page helpful to you?

[Learn how to contribute.](#)

This page was last modified on Jan 1, 2024 by [MDN contributors](#).



HTML text fundamentals

One of HTML's main jobs is to give text structure so that a browser can display an HTML document the way its developer intends. This article explains the way [HTML](#) can be used to structure a page of text by adding headings and paragraphs, emphasizing words, creating lists, and more.

Prerequisites:	Basic HTML familiarity, as covered in Getting started with HTML .
Objective:	Learn how to mark up a basic page of text to give it structure and meaning — including paragraphs, headings, lists, emphasis, and quotations.

The basics: headings and paragraphs

Most structured text consists of headings and paragraphs, whether you are reading a story, a newspaper, a college textbook, a magazine, etc.



Structured content makes the reading experience easier and more enjoyable.

In HTML, each paragraph has to be wrapped in a `<p>` element, like so:

```
HTML
<p>I am a paragraph, oh yes I am.</p>
```

Each heading has to be wrapped in a heading element:

```
HTML
<h1>I am the title of the story.</h1>
```

There are six heading elements: `h1`, `h2`, `h3`, `h4`, `h5`, and `h6`. Each element represents a different level of content in the document; `<h1>` represents the main heading, `<h2>` represents subheadings, `<h3>` represents sub-subheadings, and so on.

Implementing structural hierarchy

For example, in this story, the `<h1>` element represents the title of the story, the `<h2>` elements represent the title of each chapter, and the `<h3>` elements represent subsections of each chapter:

HTML

```
<h1>The Crushing Bore</h1>
```

```
<p>By Chris Mills</p>
```

```
<h2>Chapter 1: The dark night</h2>
```

```
<p>
  It was a dark night. Somewhere, an owl hooted. The rain lashed down on the...
</p>
```

```
<h2>Chapter 2: The eternal silence</h2>
```

```
<p>Our protagonist could not so much as a whisper out of the shadowy figure...</p>
```

```
<h3>The specter speaks</h3>
```

```
<p>
  Several more hours had passed, when all of a sudden the specter sat bolt
  upright and exclaimed, "Please have mercy on my soul!"
</p>
```

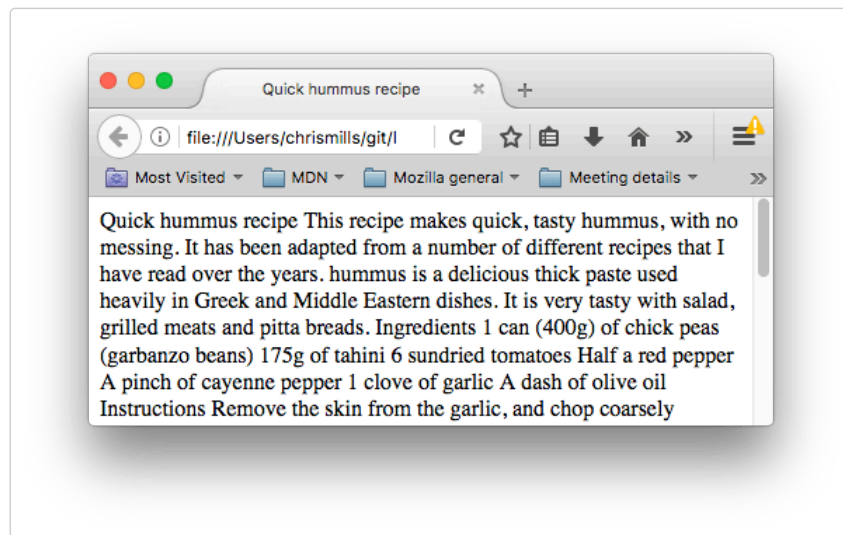
It's really up to you what the elements involved represent, as long as the hierarchy makes sense. You just need to bear in mind a few best practices as you create such structures:

- Preferably, you should use a single `<h1>` per page—this is the top level heading, and all others sit below this in the hierarchy.
- Make sure you use the headings in the correct order in the hierarchy. Don't use `<h3>` elements to represent subheadings, followed by `<h2>` elements to represent sub-subheadings—that doesn't make sense and will lead to weird results.
- Of the six heading levels available, you should aim to use no more than three per page, unless you feel it is necessary. Documents with many levels (for example, a deep heading hierarchy) become unwieldy and difficult to navigate. On such occasions, it is advisable to spread the content over multiple pages if possible.

Why do we need structure?

To answer this question, let's take a look at [text-start.html](#) —the starting point of our running example for this article (a nice hummus recipe). You should save a copy of this file on your local machine, as you'll need it for the exercises later on. This document's body currently contains multiple pieces of content. They aren't marked up in any way, but they are separated with line breaks (Enter/Return pressed to go onto the next line).

However, when you open the document in your browser, you'll see that the text appears as a big chunk!



This is because there are no elements to give the content structure, so the browser does not know what is a heading and what is a paragraph. Furthermore:

- Users looking at a web page tend to scan quickly to find relevant content, often just reading the headings, to begin with. (We usually [spend a very short time on a web page](#) .) If they can't see anything useful within a few seconds, they'll likely get frustrated and go somewhere else.
- Search engines indexing your page consider the contents of headings as important keywords for influencing the page's search rankings. Without headings, your page will perform poorly in terms of [SEO](#) (Search Engine Optimization).
- Severely visually impaired people often don't read web pages; they listen to them instead. This is done with software called a [screen reader](#) . This software provides ways to get fast access to given text content. Among the various techniques used, they provide an outline of the document by reading out the headings, allowing their users to find the information they need quickly. If headings are not available, they will be forced to listen to the whole document read out loud.
- To style content with [CSS](#), or make it do interesting things with [JavaScript](#), you need to have elements wrapping the relevant content, so CSS/JavaScript can effectively target it.

Therefore, we need to give our content structural markup.

Active learning: Giving our content structure

Let's jump straight in with a live example. In the example below, add elements to the raw text in the *Input* field so that it appears as a heading and two paragraphs in the *Output* field.

If you make a mistake, you can always reset it using the *Reset* button. If you get stuck, press the *Show solution* button to see the answer.

Why do we need semantics?

Semantics are relied on everywhere around us—we rely on previous experience to tell us what the function of an everyday object is; when we see something, we know what its function will be. So, for example, we expect a red traffic light to mean "stop," and a green traffic light to mean "go." Things can get tricky very quickly if the wrong semantics are applied. (Do any countries use red to mean "go"? We hope not.)

In a similar way, we need to make sure we are using the correct elements, giving our content the correct meaning, function, or appearance. In this context, the [h1](#) element is also a semantic element, which gives the text it wraps around the role (or meaning) of "a top level heading on your page."

HTML

```
<h1>This is a top level heading</h1>
```

By default, the browser will give it a large font size to make it look like a heading (although you could style it to look like anything you wanted using CSS). More importantly, its semantic value will be used in multiple ways, for example by search engines and screen readers (as mentioned above).

On the other hand, you could make any element *look* like a top level heading. Consider the following:

HTML

```
<span style="font-size: 32px; margin: 21px 0; display: block;">
  Is this a top level heading?
</span>
```

This is a [](#) element. It has no semantics. You use it to wrap content when you want to apply CSS to it (or do something to it with JavaScript) without giving it any extra meaning. (You'll find out more about these later on in the course.) We've applied some CSS to it to make it look like a top level heading, but since it has no semantic value, it will not get any of the extra benefits described above. It is a good idea to use the relevant HTML element for the job.

Lists

Now let's turn our attention to lists. Lists are everywhere in life—from your shopping list to the list of directions you subconsciously follow to get to your house every day, to the lists of instructions you are following in these tutorials! On the web, we have three types of lists: unordered, ordered, and description.

Unordered and ordered lists are very common, and they're covered in this section. Description lists are less common, and we'll cover them in [Advanced text formatting](#).

Unordered

Unordered lists are used to mark up lists of items for which the order of the items doesn't matter. Let's take a shopping list as an example:

```
milk
eggs
bread
hummus
```

Every unordered list starts off with a [](#) element—this wraps around all the list items:

```
HTML
<ul>
  milk
  eggs
  bread
  hummus
</ul>
```

The last step is to wrap each list item in a [](#) (list item) element:

```
HTML
<ul>
  <li>milk</li>
  <li>eggs</li>
  <li>bread</li>
  <li>hummus</li>
</ul>
```

Active learning: Marking up an unordered list

Try editing the live sample below to create your very own HTML unordered list.

Play

Ordered

Ordered lists are lists in which the order of the items *does* matter. Let's take a set of directions as an example:

Drive to the end of the road
Turn right
Go straight across the first two roundabouts
Turn left at the third roundabout
The school is on your right, 300 meters up the road

The markup structure is the same as for unordered lists, except that you have to wrap the list items in an `` element, rather than ``:

HTML

```
<ol>
  <li>Drive to the end of the road</li>
  <li>Turn right</li>
  <li>Go straight across the first two roundabouts</li>
  <li>Turn left at the third roundabout</li>
  <li>The school is on your right, 300 meters up the road</li>
</ol>
```

Active learning: Marking up an ordered list

Try editing the live sample below to create your very own HTML ordered list.

Play

Active learning: Marking up our recipe page

So at this point in the article, you have all the information you need to mark up our recipe page example. You can choose to either save a local copy of our [text-start.html](#) starting file and do the work there or do it in the editable example below. Doing it locally will probably be better, as then you'll get to save the work you are doing, whereas if you fill it in to the editable example, it will be lost the next time you open the page. Both have pros and cons.

Play

If you get stuck, you can always press the *Show solution* button, or check out our [text-complete.html](#) example on our GitHub repo.

Nesting lists

It is perfectly OK to nest one list inside another one. You might want to have some sub-bullets sitting below a top-level bullet. Let's take the second list from our recipe example:

HTML

```
<ol>
  <li>Remove the skin from the garlic, and chop coarsely.</li>
  <li>Remove all the seeds and stalk from the pepper, and chop coarsely.</li>
  <li>Add all the ingredients into a food processor.</li>
  <li>Process all the ingredients into a paste.</li>
  <li>If you want a coarse "chunky" hummus, process it for a short time.</li>
  <li>If you want a smooth hummus, process it for a longer time.</li>
</ol>
```

Since the last two bullets are very closely related to the one before them (they read like sub-instructions or choices that fit below that bullet), it might make sense to nest them inside their own unordered list and put that list inside the current fourth bullet. This would look like so:

HTML

```
<ol>
  <li>Remove the skin from the garlic, and chop coarsely.</li>
  <li>Remove all the seeds and stalk from the pepper, and chop coarsely.</li>
  <li>Add all the ingredients into a food processor.</li>
  <li>
    Process all the ingredients into a paste.
    <ul>
      <li>
```

```
        If you want a coarse "chunky" hummus, process it for a short time.
    </li>
    <li>If you want a smooth hummus, process it for a longer time.</li>
</ul>
</li>
</ol>
```

Try going back to the previous active learning example and updating the second list like this.

Emphasis and importance

In human language, we often emphasize certain words to alter the meaning of a sentence, and we often want to mark certain words as important or different in some way. HTML provides various semantic elements to allow us to mark up textual content with such effects, and in this section, we'll look at a few of the most common ones.

Emphasis

When we want to add emphasis in spoken language, we *stress* certain words, subtly altering the meaning of what we are saying. Similarly, in written language we tend to stress words by putting them in italics. For example, the following two sentences have different meanings.


I am glad you weren't late.

I am *glad* you weren't *late*.

The first sentence sounds genuinely relieved that the person wasn't late. In contrast, the second one, with both the words "glad" and "late" in italics, sounds sarcastic or passive-aggressive, expressing annoyance that the person arrived a bit late.

In HTML we use the [](#) (emphasis) element to mark up such instances. As well as making the document more interesting to read, these are recognized by screen readers, which can be configured to speak them in a different tone of voice. Browsers style this as italic by default, but you shouldn't use this tag purely to get italic styling. To do that, you'd use a [](#) element and some CSS, or perhaps an [<i>](#) element (see below).

HTML

 mdn web docs [_](#)

Strong importance

To emphasize important words, we tend to stress them in spoken language and **bold** them in written language. For example:

This liquid is **highly toxic**.

I am counting on you. **Do not** be late!

In HTML we use the [](#) (strong importance) element to mark up such instances. As well as making the document more useful, again these are recognized by screen readers, which can be configured to speak them in a different tone of voice. Browsers style this as bold text by default, but you shouldn't use this tag purely to get bold styling. To do that, you'd use a [](#) element and some CSS, or perhaps a [](#) element (see below).

HTML

Play

```
<p>This liquid is <strong>highly toxic</strong>.</p>
```

```
<p>I am counting on you. <strong>Do not</strong> be late!</p>
```

You can nest strong and emphasis inside one another if desired:

HTML

Play

```
<p>This liquid is <strong>highly toxic</strong> – if you drink it, <strong>you may <em>die</em></strong>.</p>
```

Play

Active learning: Let's be important

In this active learning section, we've provided an editable example. Inside it, we'd like you to try adding emphasis and strong importance to the words you think need them, just to have some practice.

Play

Italic, bold, underline...

The elements we've discussed so far have clear-cut associated semantics. The situation with [](#), [<i>](#), and [<u>](#) is somewhat more complicated. They came about so people could write bold, italics, or underlined text in an era when CSS was still supported poorly or not at all. Elements like this, which only affect presentation and not semantics, are known as **presentational elements** and should no longer be used because, as we've seen before, semantics is so important to accessibility, SEO, etc.

HTML5 redefined ``, `<i>`, and `<u>` with new, somewhat confusing, semantic roles.

Here's the best rule you can remember: It's only appropriate to use ``, `<i>`, or `<u>` to convey a meaning traditionally conveyed with bold, italics, or underline when there isn't a more suitable element; and there usually is. Consider whether ``, ``, `<mark>`, or `` might be more appropriate.

Always keep an accessibility mindset. The concept of italics isn't very helpful to people using screen readers, or to people using a writing system other than the Latin alphabet.

- [<i>](#) is used to convey a meaning traditionally conveyed by italic: foreign words, taxonomic designation, technical terms, a thought...
- [](#) is used to convey a meaning traditionally conveyed by bold: keywords, product names, lead sentence...
- [<u>](#) is used to convey a meaning traditionally conveyed by underline: proper name, misspelling...

Note: People strongly associate underlining with hyperlinks. Therefore, on the web, it's best to only underline links. Use the `<u>` element when it's semantically appropriate, but consider using CSS to change the default underline to something more appropriate on the web. The example below illustrates how it can be done.

HTML

Play

```
<!-- scientific names -->
<p>
  The Ruby-throated Hummingbird (<i>Archilochus colubris</i>) is the most common
  hummingbird in Eastern North America.
</p>

<!-- foreign words -->
<p>
  The menu was a sea of exotic words like <i lang="uk-latn">vatrushka</i>,
  <i lang="id">nasi goreng</i> and <i lang="fr">soupe à l'oignon</i>.
</p>

<!-- a known misspelling -->
<p>Someday I'll learn how to <u class="spelling-error">spel</u> better.</p>

<!-- term being defined when used in a definition -->
<dl>
  <dt>Semantic HTML</dt>
  <dd>
    Use the elements based on their <b>semantic</b> meaning, not their
    appearance.
  </dd>
</dl>
```

Play

Test your skills!

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

Summary

That's it for now! This article should have given you a good idea of how to start marking up text in HTML and introduced you to some of the most important elements in this area. There are a lot more semantic elements to cover in this area, and we'll look at a lot more in our [Advanced text formatting](#) article later on in the course. In the next article, we'll be looking in detail at how to [create hyperlinks](#), possibly the most important element on the web.

Help improve MDN

Was this page helpful to you?

[Learn how to contribute.](#)

This page was last modified on Mar 13, 2024 by [MDN contributors](#).



Creating hyperlinks

Hyperlinks are really important — they are what makes the Web *a web*. This article shows the syntax required to make a link, and discusses link best practices.

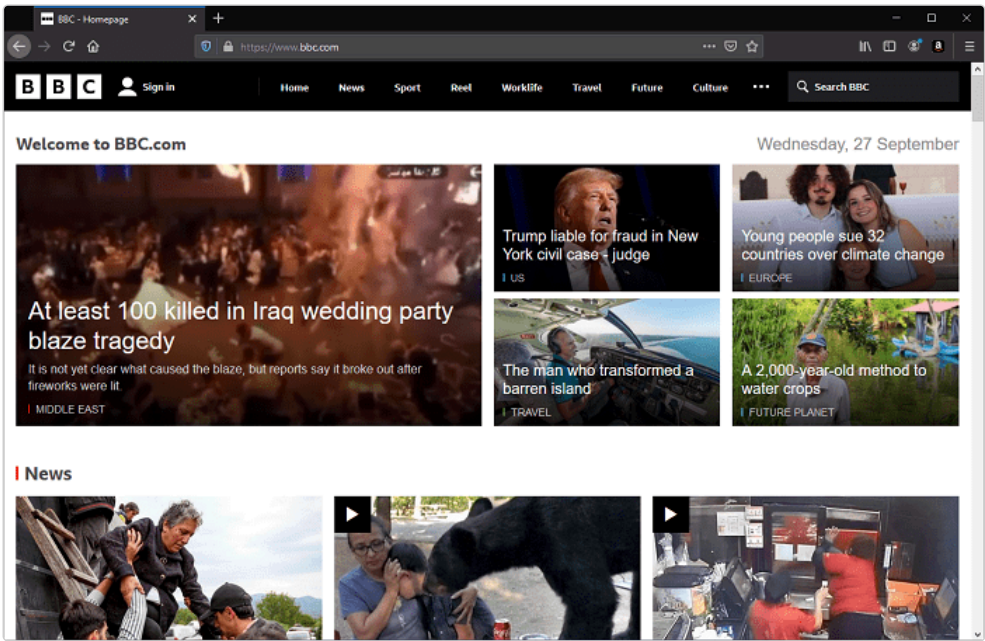
Prerequisites:	Basic HTML familiarity, as covered in Getting started with HTML . HTML text formatting, as covered in HTML text fundamentals .
Objective:	To learn how to implement a hyperlink effectively, and link multiple files together.

What is a hyperlink?

Hyperlinks are one of the most exciting innovations the Web has to offer. They've been a feature of the Web since the beginning, and are what makes the Web *a web*. Hyperlinks allow us to link documents to other documents or resources, link to specific parts of documents, or make apps available at a web address. Almost any web content can be converted to a link so that when clicked or otherwise activated the web browser goes to another web address ([URL](#)).

Note: A URL can point to HTML files, text files, images, text documents, video and audio files, or anything else that lives on the Web. If the web browser doesn't know how to display or handle the file, it will ask you if you want to open the file (in which case the duty of opening or handling the file is passed to a suitable native app on the device) or download the file (in which case you can try to deal with it later on).

For example, the BBC homepage contains many links that point not only to multiple news stories, but also different areas of the site (navigation functionality), login/registration pages (user tools), and more.



Anatomy of a link

A basic link is created by wrapping the text or other content inside an `<a>` element and using the `href` attribute, also known as a **Hypertext Reference**, or **target**, that contains the web address.

HTML

```
<p>
  I'm creating a link to
  <a href="https://www.mozilla.org/en-US/">the Mozilla homepage</a>.
</p>
```

This gives us the following result:

I'm creating a link to [the Mozilla homepage](https://www.mozilla.org/en-US/) .

Block level links

As mentioned before, almost any content can be made into a link, even [block-level elements](#). If you want to make a heading element a link then wrap it in an anchor (`<a>`) element as shown in the following code snippet:

HTML

Play

```
<a href="https://developer.mozilla.org/en-US/">
  <h1>MDN Web Docs</h1>
</a>
<p>
  Documenting web technologies, including CSS, HTML, and JavaScript, since 2005.
</p>
```

This turns the heading into a link:

Play

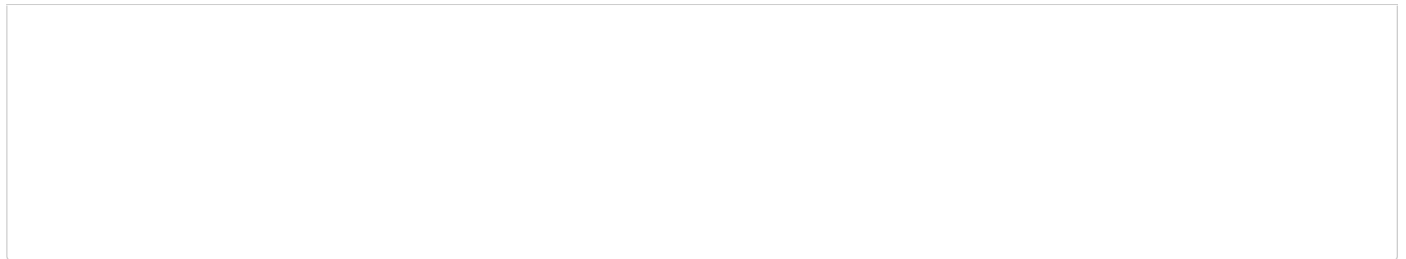


Image links

If you have an image you want to make into a link, use the `<a>` element to wrap the image file referenced with the `` element. The example below uses a relative path to reference a locally stored SVG image file.

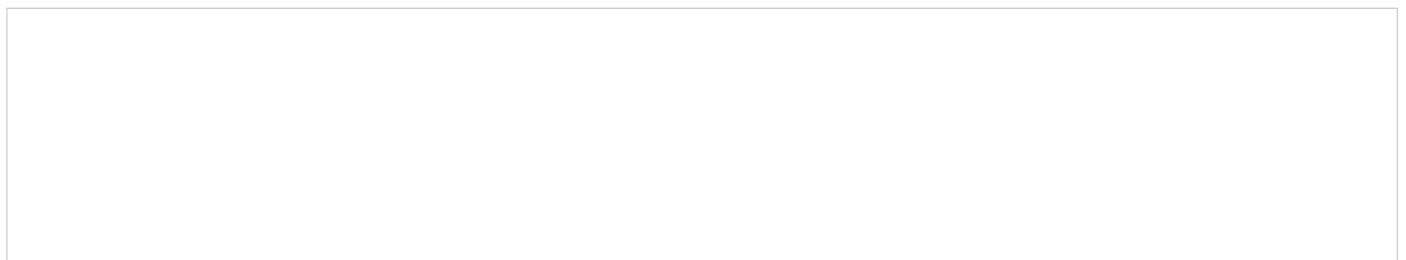
HTML

Play

```
<a href="https://developer.mozilla.org/en-US/">
  
</a>
```

This makes the MDN logo a link:

Play



Note: You'll find out more about using images on the Web in a future article.

Adding supporting information with the title attribute

Another attribute you may want to add to your links is `title`. The title contains additional information about the link, such as which kind of information the page contains, or things to be aware of on the website.

HTML Play

```
<p>
  I'm creating a link to
  <a
    href="https://www.mozilla.org/en-US/"
    title="The best place to find more information about Mozilla's
      mission and how to contribute">
    the Mozilla homepage</a>.
</p>
```

This gives us the following result and hovering over the link displays the title as a tooltip:

Play

Note: A link title is only revealed on mouse hover, which means that people relying on keyboard controls or touchscreens to navigate web pages will have difficulty accessing title information. If a title's information is truly important to the usability of the page, then you should present it in a manner that will be accessible to all users, for example by putting it in the regular text.

Active learning: creating your own example link

Create an HTML document using your local code editor and our [getting started template](#).

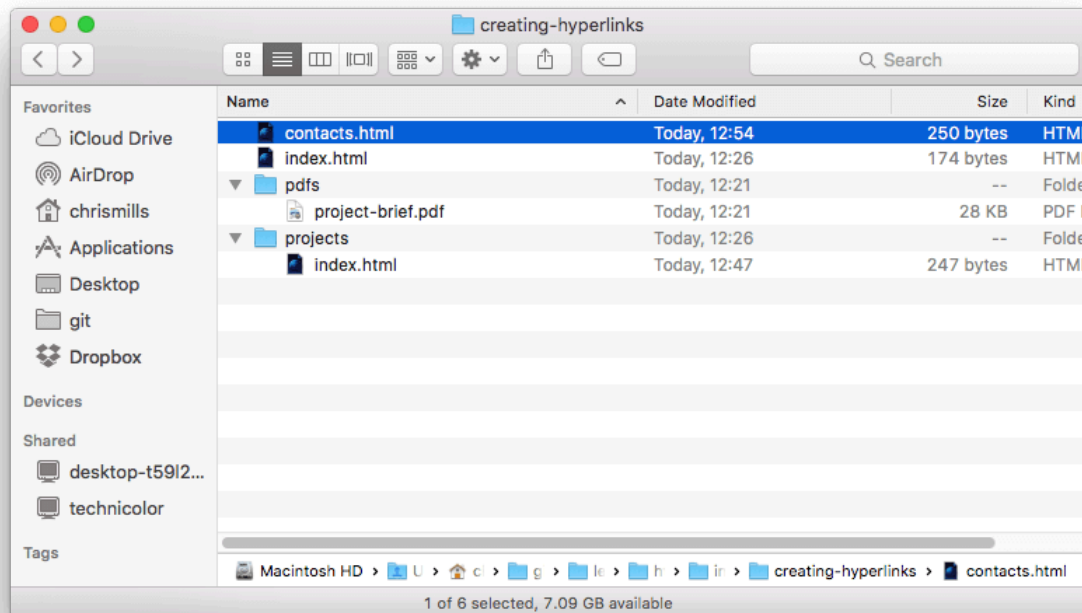
- Inside the HTML body, add one or more paragraphs or other types of content you already know about.
- Change some of the content into links.
- Include title attributes.

A quick primer on URLs and paths

To fully understand link targets, you need to understand URLs and file paths. This section gives you the information you need to achieve this.

A URL, or Uniform Resource Locator is a string of text that defines where something is located on the Web. For example, Mozilla's English homepage is located at `https://www.mozilla.org/en-US/`.

URLs use paths to find files. Paths specify where the file you're interested in is located in the filesystem. Let's look at an example of a directory structure, see the [creating hyperlinks](#) directory.



The **root** of this directory structure is called `creating-hyperlinks`. When working locally with a website, you'll have one directory that contains the entire site. Inside the **root**, we have an `index.html` file and a `contacts.html`. In a real website, `index.html` would be our home page or landing page (a web page that serves as the entry point for a website or a particular section of a website.).

There are also two directories inside our root — `pdfs` and `projects`. These each have a single file inside them — a PDF (`project-brief.pdf`) and an `index.html` file, respectively. Note that you can have two `index.html` files in one project, as long as they're in different filesystem locations. The second `index.html` would perhaps be the main landing page for project-related information.

- **Same directory:** If you wanted to include a hyperlink inside `index.html` (the top level `index.html`) pointing to `contacts.html`, you would specify the filename that you want to link to, because it's in the same directory as the current file. The URL you would use is `contacts.html`:

HTML

```
<p>
  Want to contact a specific staff member? Find details on our
  <a href="contacts.html">contacts page</a>.
</p>
```

- **Moving down into subdirectories:** If you wanted to include a hyperlink inside `index.html` (the top level `index.html`) pointing to `projects/index.html`, you would need to go down into the `projects` directory before indicating the file you want to link to. This is done by specifying the directory's name, then a forward slash, then the name of the file. The URL you would use is `projects/index.html`:

HTML

```
<p>Visit my <a href="projects/index.html">project homepage</a>.</p>
```

Moving back up into parent directories: If you wanted to include a hyperlink inside `contacts.html` pointing to

[mdn web docs](#)

HTML

```
<p>A link to my <a href="../pdfs/project-brief.pdf">project brief</a>.</p>
```

Note: You can combine multiple instances of these features into complex URLs, if needed, for example:
../../../../complex/path/to/my/file.html .

Document fragments

It's possible to link to a specific part of an HTML document, known as a **document fragment**, rather than just to the top of the document. To do this you first have to assign an [id](#) attribute to the element you want to link to. It normally makes sense to link to a specific heading, so this would look something like the following:

HTML

```
<h2 id="Mailing_address">Mailing address</h2>
```

Then to link to that specific `id`, you'd include it at the end of the URL, preceded by a hash/pound symbol (`#`), for example:

HTML

```
<p>
  Want to write us a letter? Use our
  <a href="contacts.html#Mailing_address">mailing address</a>.
</p>
```

You can even use the document fragment reference on its own to link to *another part of the current document*:

HTML

```
<p>
  The <a href="#Mailing_address">company mailing address</a> can be found at the
  bottom of this page.
</p>
```

Absolute versus relative URLs

Two terms you'll come across on the Web are **absolute URL** and **relative URL**:

absolute URL: Points to a location defined by its absolute location on the web, including [protocol](#) and [domain name](#). For example, if an `index.html` page is uploaded to a directory called `projects` that sits inside the **root** of a web server, and the website's domain is `https://www.example.com`, the page would be available at `https://www.example.com/projects/index.html` (or even just `https://www.example.com/projects/`, as most web servers just look for a landing page such as `index.html` to load if it isn't specified in the URL.)

An absolute URL will always point to the same location, no matter where it's used.

relative URL: Points to a location that is *relative* to the file you are linking from, more like what we looked at in the previous section. For example, if we wanted to link from our example file at `https://www.example.com/projects/index.html` to a PDF file in the same directory, the URL would just be the filename — `project-brief.pdf` — no extra information needed. If the PDF was available in a subdirectory inside `projects` called `pdfs`, the relative link would be `pdfs/project-brief.pdf` (the equivalent absolute URL would be `https://www.example.com/projects/pdfs/project-brief.pdf`.)

A relative URL will point to different places depending on the actual location of the file you refer from — for example if we moved our `index.html` file out of the `projects` directory and into the **root** of the website (the top level, not in any directories), the `pdfs/project-brief.pdf` relative URL link inside it would now point to a file located at `https://www.example.com/pdfs/project-brief.pdf`, not a file located at `https://www.example.com/projects/pdfs/project-brief.pdf`.

Of course, the location of the `project-brief.pdf` file and `pdfs` folder won't suddenly change because you moved the `index.html` file — this would make your link point to the wrong place, so it wouldn't work if clicked on. You need to be careful!

Link best practices

There are some best practices to follow when writing links. Let's look at these now.

Use clear link wording

It's easy to throw links up on your page. That's not enough. We need to make our links *accessible* to all readers, regardless of their current context and which tools they prefer. For example:

- Screen reader users like jumping around from link to link on the page, and reading links out of context.
- Search engines use link text to index target files, so it is a good idea to include keywords in your link text to effectively describe what is being linked to.
- Visual readers skim over the page rather than reading every word, and their eyes will be drawn to page features that stand out, like links. They will find descriptive link text useful.

Let's look at a specific example:

Good link text: [Download Firefox](https://www.mozilla.org/firefox/)

HTML

```
<p><a href="https://www.mozilla.org/firefox/">Download Firefox</a></p>
```

Bad link text: [Click here](https://www.mozilla.org/firefox/) to download Firefox

HTML

```
<p>  
  <a href="https://www.mozilla.org/firefox/">Click here</a> to download Firefox  
</p>
```

Other tips:

- Don't repeat the URL as part of the link text — URLs look ugly, and sound even uglier when a screen reader reads them out letter by letter.
- Don't say "link" or "links to" in the link text — it's just noise. Screen readers tell people there's a link. Visual users will also know there's a link, because links are generally styled in a different color and underlined (this convention generally shouldn't be broken, as users are used to it).
- Keep your link text as short as possible — this is helpful because screen readers need to interpret the entire link text.
- Minimize instances where multiple copies of the same text are linked to different places. This can cause problems for screen reader users, if there's a list of links out of context that are labeled "click here", "click here", "click here".

Linking to non-HTML resources — leave clear signposts

When linking to a resource that will be downloaded (like a PDF or Word document), streamed (like video or audio), or has another potentially unexpected effect (opens a popup window), you should add clear wording to reduce any confusion.

For example:

- If you're on a low bandwidth connection, click a link, and then a multiple megabyte download starts unexpectedly.

Let's look at some examples, to see what kind of text can be used here:

HTML

```
<p>
  <a href="https://www.example.com/large-report.pdf">
    Download the sales report (PDF, 10MB)
  </a>
</p>

<p>
  <a href="https://www.example.com/video-stream/" target="_blank">
    Watch the video (stream opens in separate tab, HD quality)
  </a>
</p>
```

Use the download attribute when linking to a download

When you are linking to a resource that's to be downloaded rather than opened in the browser, you can use the `download` attribute to provide a default save filename. Here's an example with a download link to the latest Windows version of Firefox:

HTML

```
<a
  href="https://download.mozilla.org/?product=firefox-latest-ssl&os=win64&lang=en-US"
  download="firefox-latest-64bit-installer.exe">
  Download Latest Firefox for Windows (64-bit) (English, US)
</a>
```

Active learning: creating a navigation menu

For this exercise, we'd like you to link some pages together with a navigation menu to create a multipage website. This is one common way in which a website is created — the same page structure is used on every page, including the same navigation menu, so when links are clicked it gives the impression that you are staying in the same place, and different content is being brought up.

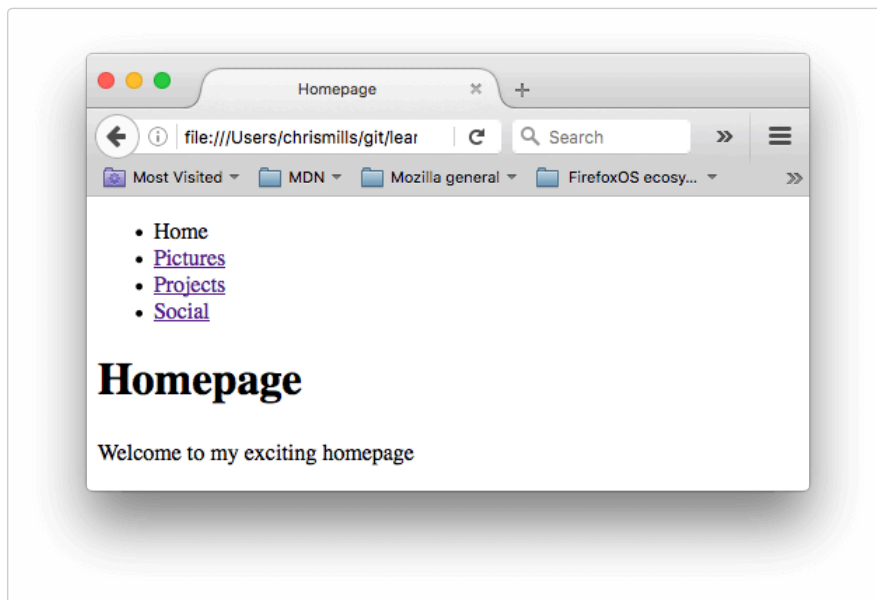
You'll need to make local copies of the following four pages, all in the same directory. For a complete file list, see the [navigation-menu-start](#) directory:

- [index.html](#)
- [projects.html](#)
- [pictures.html](#)
- [social.html](#)

You should:

1. Add an unordered list in the indicated place on one page that includes the names of the pages to link to. A navigation menu is usually just a list of links, so this is semantically OK.
2. Change each page name into a link to that page.
3. Copy the navigation menu across to each page.
4. On each page, remove just the link to that same page — it's confusing and unnecessary for a page to include a link to itself. And, the lack of a link acts a good visual reminder of which page you are currently on.

The finished example should look similar to the following page:



Note: If you get stuck, or aren't sure if you have got it right, you can check the [navigation-menu-marked-up](#) directory to see the correct answer.

Email links

It's possible to create links or buttons that, when clicked, open a new outgoing email message rather than linking to a resource or page. This is done using the `<a>` element and the `mailto:` URL scheme.

In its most basic and commonly used form, a `mailto:` link indicates the email address of the intended recipient. For example:

HTML

```
<a href="mailto:nowhere@mozilla.org">Send email to nowhere</a>
```

This results in a link that looks like this: [Send email to nowhere](mailto:nowhere@mozilla.org).

In fact, the email address is optional. If you omit it and your `href` is "mailto:", a new outgoing email window will be opened by the user's email client with no destination address. This is often useful as "Share" links that users can click to send an email to an address of their choosing.

Specifying details

In addition to the email address, you can provide other information. In fact, any standard mail header fields can be added to the `mailto` URL you provide. The most commonly used of these are "subject", "cc", and "body" (which is not a true header field, but allows you to specify a short content message for the new email). Each field and its value is specified as a query term.

Here's an example that includes a cc, bcc, subject and body:

HTML

```
<a
  href="mailto:nowhere@mozilla.org?
cc=name2@rapidtables.com&bcc=name3@rapidtables.com&subject=The%20subject%20of%20the%20email&body=The%20body%20of%20the%20email"
  >
  Send mail with cc, bcc, subject and body
</a>
```

Note: The values of each field must be URL-encoded with non-printing characters (invisible characters like tabs, carriage returns, and page breaks) and spaces percent-escaped . Also, note the use of the question mark (?) to separate the main URL from the field values, and ampersands (&) to separate each field in the `mailto:` URL. This is standard URL query notation. Read The GET method to understand what URL query notation is more commonly used for.

Here are a few other sample `mailto` URLs:

- <mailto:>
- <mailto:nowhere@mozilla.org>
- <mailto:nowhere@mozilla.org,nobody@mozilla.org>
- <mailto:nowhere@mozilla.org?cc=nobody@mozilla.org>
- <mailto:nowhere@mozilla.org?cc=nobody@mozilla.org&subject=This%20is%20the%20subject>

Test your skills!

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

Summary

That's it for links, for now anyway! You'll return to links later on in the course when you start to look at styling them. Next up for HTML, we'll return to text semantics and look at some more advanced/unusual features that you'll find useful — [Advanced text formatting](#) is your next stop.

Help improve MDN

Was this page helpful to you?

Yes

No

[Learn how to contribute.](#)

This page was last modified on Nov 2, 2023 by [MDN contributors](#).



Document and website structure

In addition to defining individual parts of your page (such as "a paragraph" or "an image"), [HTML](#) also boasts a number of block level elements used to define areas of your website (such as "the header", "the navigation menu", "the main content column"). This article looks into how to plan a basic website structure, and write the HTML to represent this structure.

Prerequisites:	Basic HTML familiarity, as covered in Getting started with HTML . HTML text formatting, as covered in HTML text fundamentals . How hyperlinks work, as covered in Creating hyperlinks .
Objective:	Learn how to structure your document using semantic tags, and how to work out the structure of a simple website.

Basic sections of a document

Webpages can and will look pretty different from one another, but they all tend to share similar standard components, unless the page is displaying a fullscreen video or game, is part of some kind of art project, or is just badly structured:

header:

Usually a big strip across the top with a big heading, logo, and perhaps a tagline. This usually stays the same from one webpage to another.

navigation bar:

Links to the site's main sections; usually represented by menu buttons, links, or tabs. Like the header, this content usually remains consistent from one webpage to another — having inconsistent navigation on your website will just lead to confused, frustrated users. Many web designers consider the navigation bar to be part of the header rather than an individual component, but that's not a requirement; in fact, some also argue that having the two separate is better for [accessibility](#), as screen readers can read the two features better if they are separate.

main content:

A big area in the center that contains most of the unique content of a given webpage, for example, the video you want to watch, or the main story you're reading, or the map you want to view, or the news headlines, etc. This is the one part of the website that definitely will vary from page to page!

sidebar:

Some peripheral info, links, quotes, ads, etc. Usually, this is contextual to what is contained in the main content (for example on a news article page, the sidebar might contain the author's bio, or links to related articles) but there are also cases where you'll find some recurring elements like a secondary navigation system.

footer:

A strip across the bottom of the page that generally contains fine print, copyright notices, or contact info. It's a place to put common information (like the header) but usually, that information is not critical or secondary to the website itself. The footer is also sometimes used for [SEO](#) purposes, by providing links for quick access to popular content.

A "typical website" could be structured something like this:

Header



Note: The image above illustrates the main sections of a document, which you can define with HTML. However, the *appearance* of the page shown here - including the layout, colors, and fonts - is achieved by applying [CSS](#) to the HTML.

In this module we're not teaching CSS, but once you have an understanding of the basics of HTML, try diving into our [CSS first steps](#) module to start learning how to style your site.

HTML for structuring content

The simple example shown above isn't pretty, but it is perfectly fine for illustrating a typical website layout example. Some websites have more columns, some are a lot more complex, but you get the idea. With the right CSS, you could use pretty much any elements to wrap around the different sections and get it looking how you wanted, but as discussed before, we need to respect semantics and **use the right element for the right job**.

This is because visuals don't tell the whole story. We use color and font size to draw sighted users' attention to the most useful parts of the content, like the navigation menu and related links, but what about visually impaired people for example, who might not find concepts like "pink" and "large font" very useful?

Note: [Roughly 8% of men and 0.5% of women](#) are colorblind; or, to put it another way, approximately 1 in every 12 men and 1 in every 200 women. Blind and visually impaired people represent roughly 4-5% of the world population (in 2015 there were [940 million people with some degree of vision loss](#) , while the total population was [around 7.5 billion](#)).

In your HTML code, you can mark up sections of content based on their *functionality* — you can use elements that represent the sections of content described above unambiguously, and assistive technologies like screen readers can recognize those elements and help with tasks like "find the main navigation", or "find the main content." As we mentioned earlier in the course, there are a number of [consequences of not using the right element structure and semantics for the right job](#).

To implement such semantic mark up, HTML provides dedicated tags that you can use to represent such sections, for example:

- **header:** `<header>` .
- **navigation bar:** `<nav>` .
- **main content:** `<main>` , with various content subsections represented by `<article>` , `<section>` , and `<div>` elements.
- **sidebar:** `<aside>` ; often placed inside `<main>` .
- **footer:** `<footer>` .

Active learning: exploring the code for our example

Our example seen above is represented by the following code (you can also [find the example in our GitHub repository](#)). We'd like you to look at the example above, and then look over the listing below to see what parts make up what section of the visual.

HTML

```
<!doctype html>
<html lang="en-US">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />

    <title>My page title</title>
    <link
      href="https://fonts.googleapis.com/css?family=Open+Sans+Condensed:300|Sonsie+One"
      rel="stylesheet" />
    <link rel="stylesheet" href="style.css" />
  </head>

  <body>
    <!-- Here is our main header that is used across all the pages of our website -->

    <header>
      <h1>Header</h1>
    </header>

    <nav>
      <ul>
        <li><a href="#">Home</a></li>
        <li><a href="#">Our team</a></li>
        <li><a href="#">Projects</a></li>
        <li><a href="#">Contact</a></li>
      </ul>

      <!-- A Search form is another common non-linear way to navigate through a website. -->

      <form>
        <input type="search" name="q" placeholder="Search query" />
        <input type="submit" value="Go!" />
      </form>
    </nav>

    <!-- Here is our page's main content -->
    <main>
      <!-- It contains an article -->
      <article>
        <h2>Article heading</h2>

        <p>
          Lorem ipsum dolor sit amet, consectetur adipisicing elit. Donec a diam
          lectus. Set sit amet ipsum mauris. Maecenas congue ligula as quam
```

```

    viverra nec consectetur ant hendrerit. Donec et mollis dolor. Praesent
    et diam eget libero egestas mattis sit amet vitae augue. Nam tincidunt
    congue enim, ut porta lorem lacinia consectetur.
</p>

<h3>Subsection</h3>

<p>
    Donec ut libero sed accu vehicula ultricies a non tortor. Lorem ipsum
    dolor sit amet, consectetur adipiscing elit. Aenean ut gravida lorem.
    Ut turpis felis, pulvinar a semper sed, adipiscing id dolor.
</p>

<p>
    Pelientesque auctor nisi id magna consequat sagittis. Curabitur
    dapibus, enim sit amet elit pharetra tincidunt feugiat nist imperdiet.
    Ut convallis libero in urna ultrices accumsan. Donec sed odio eros.
</p>

<h3>Another subsection</h3>

<p>
    Donec viverra mi quis quam pulvinar at malesuada arcu rhoncus. Cum
    sociis natoque penatibus et manis dis parturient montes, nascetur
    ridiculus mus. In rutrum accumsan ultricies. Mauris vitae nisi at sem
    facilisis semper ac in est.
</p>

<p>
    Vivamus fermentum semper porta. Nunc diam velit, adipiscing ut
    tristique vitae sagittis vel odio. Maecenas convallis ullamcorper
    ultricies. Curabitur ornare, ligula semper consectetur sagittis, nisi
    diam iaculis velit, is fringille sem nunc vet mi.
</p>
</article>

<!-- the aside content can also be nested within the main content -->
<aside>
    <h2>Related</h2>

    <ul>
        <li><a href="#">Oh I do like to be beside the seaside</a></li>
        <li><a href="#">Oh I do like to be beside the sea</a></li>
        <li><a href="#">Although in the North of England</a></li>
        <li><a href="#">It never stops raining</a></li>
        <li><a href="#">Oh well...</a></li>
    </ul>
</aside>
</main>

<!-- And here is our main footer that is used across all the pages of our website -->

<footer>
    <p>©Copyright 2050 by nobody. All rights reversed.</p>
</footer>
</body>
</html>

```

Take some time to look over the code and understand it — the comments inside the code should also help you to understand it. We aren't asking you to do much else in this article, because the key to understanding document layout is writing a sound HTML structure, and then laying it out with CSS. We'll wait for this until you start to study CSS layout as part of the CSS topic.

HTML layout elements in more detail

It's good to understand the overall meaning of all the HTML sectioning elements in detail — this is something you'll work on gradually as you start to get more experience with web development. You can find a lot of detail by reading our [HTML element reference](#). For now, these are the main definitions that you should try to understand:

- [<main>](#) is for content *unique to this page*. Use `<main>` only *once* per page, and put it directly inside [<body>](#). Ideally this shouldn't be nested within other elements.
- [<article>](#) encloses a block of related content that makes sense on its own without the rest of the page (e.g., a single blog post).
- [<section>](#) is similar to `<article>`, but it is more for grouping together a single part of the page that constitutes one single piece of functionality (e.g., a mini map, or a set of article headlines and summaries), or a theme. It's considered best practice to begin each section with a [heading](#); also note that you can break `<article>`s up into different `<section>`s, or `<section>`s up into different `<article>`s, depending on the context.
- [<aside>](#) contains content that is not directly related to the main content but can provide additional information indirectly related to it (glossary entries, author biography, related links, etc.).
- [<header>](#) represents a group of introductory content. If it is a child of [<body>](#) it defines the global header of a webpage, but if it's a child of an [<article>](#) or [<section>](#) it defines a specific header for that section (try not to confuse this with [titles and headings](#)).
- [<nav>](#) contains the main navigation functionality for the page. Secondary links, etc., would not go in the navigation.
- [<footer>](#) represents a group of end content for a page.

Each of the aforementioned elements can be clicked on to read the corresponding article in the "HTML element reference" section, providing more detail about each one.

Non-semantic wrappers

Sometimes you'll come across a situation where you can't find an ideal semantic element to group some items together or wrap some content. Sometimes you might want to just group a set of elements together to affect them all as a single entity with some [CSS](#) or [JavaScript](#). For cases like these, HTML provides the [<div>](#) and [](#) elements. You should use these preferably with a suitable [class](#) attribute, to provide some kind of label for them so they can be easily targeted.

[](#) is an inline non-semantic element, which you should only use if you can't think of a better semantic text element to wrap your content, or don't want to add any specific meaning. For example:

```
HTML
<p>
  The King walked drunkenly back to his room at 01:00, the beer doing nothing to
  aid him as he staggered through the door.
  <span class="editor-note">
    [Editor's note: At this point in the play, the lights should be down low].
  </span>
</p>
```

In this case, the editor's note is supposed to merely provide extra direction for the director of the play; it is not supposed to have extra semantic meaning. For sighted users, CSS would perhaps be used to distance the note slightly from the main text.

[<div>](#) is a block level non-semantic element, which you should only use if you can't think of a better semantic block element to use, or don't want to add any specific meaning. For example, imagine a shopping cart widget that you could choose to pull up at any point during your time on an e-commerce site:

```

<div class="shopping-cart">
  <h2>Shopping cart</h2>
  <ul>
    <li>
      <p>
        <a href=""><strong>Silver earrings</strong></a>: $99.95.
      </p>
      
    </li>
    <li>...</li>
  </ul>
  <p>Total cost: $237.89</p>
</div>

```

This isn't really an `<aside>`, as it doesn't necessarily relate to the main content of the page (you want it viewable from anywhere). It doesn't even particularly warrant using a `<section>`, as it isn't part of the main content of the page. So a `<div>` is fine in this case. We've included a heading as a signpost to aid screen reader users in finding it.

Warning: Dives are so convenient to use that it's easy to use them too much. As they carry no semantic value, they just clutter your HTML code. Take care to use them only when there is no better semantic solution and try to reduce their usage to the minimum otherwise you'll have a hard time updating and maintaining your documents.

Line breaks and horizontal rules

Two elements that you'll use occasionally and will want to know about are `
` and `<hr>`.

`
`: the line break element

`
` creates a line break in a paragraph; it is the only way to force a rigid structure in a situation where you want a series of fixed short lines, such as in a postal address or a poem. For example:

HTML

Play

```

<p>
  There once was a man named O'Dell<br />
  Who loved to write HTML<br />
  But his structure was bad, his semantics were sad<br />
  and his markup didn't read very well.
</p>

```

Without the `
` elements, the paragraph would just be rendered in one long line (as we said earlier in the course, [HTML ignores most whitespace](#)); with `
` elements in the code, the markup renders like this:

Play

```

There once was a man named O'Dell
Who loved to write HTML
But his structure was bad, his semantics were sad
and his markup didn't read very well.

```

`<hr>`: the thematic break element

`<hr>` elements create a horizontal rule in the document that denotes a thematic change in the text (such as a change in topic or scene). Visually it just looks like a horizontal line. As an example:

```
<p>
  Ron was backed into a corner by the marauding netherbeasts. Scared, but
  determined to protect his friends, he raised his wand and prepared to do
  battle, hoping that his distress call had made it through.
</p>
<hr />
<p>
  Meanwhile, Harry was sitting at home, staring at his royalty statement and
  pondering when the next spin off series would come out, when an enchanted
  distress letter flew through his window and landed in his lap. He read it
  hazily and sighed; "better get back to work then", he mused.
</p>
```

Would render like this:

Play

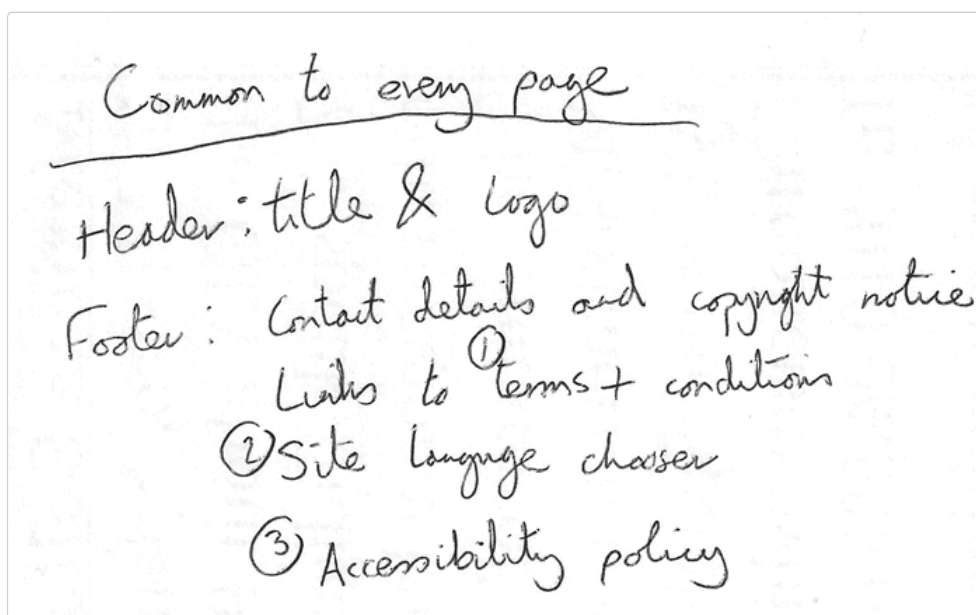
Ron was backed into a corner by the marauding netherbeasts. Scared, but determined to protect his friends, he raised his wand and prepared to do battle, hoping that his distress call had made it through.

Meanwhile, Harry was sitting at home, staring at his royalty statement and pondering when the next spin off series would come out, when an enchanted distress letter flew through his window and landed in his lap. He read it hazily and sighed; "better get back to work then", he mused.

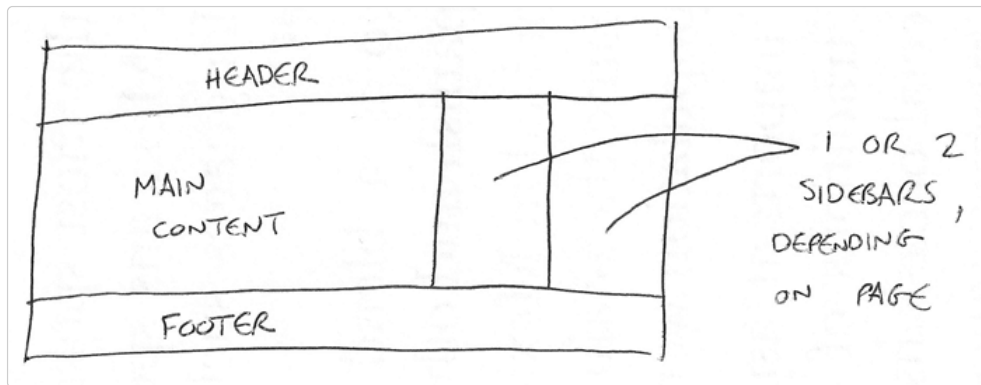
Planning a simple website

Once you've planned out the structure of a simple webpage, the next logical step is to try to work out what content you want to put on a whole website, what pages you need, and how they should be arranged and link to one another for the best possible user experience. This is called [Information architecture](#). In a large, complex website, a lot of planning can go into this process, but for a simple website of a few pages, this can be fairly simple, and fun!

1. Bear in mind that you'll have a few elements common to most (if not all) pages — such as the navigation menu, and the footer content. If your site is for a business, for example, it's a good idea to have your contact information available in the footer on each page. Note down what you want to have common to every page.



2. Next, draw a rough sketch of what you might want the structure of each page to look like (it might look like our simple website above). Note what each block is going to be.



3. Now, brainstorm all the other (not common to every page) content you want to have on your website — write a big list down.

Search for Flights
Hotels / other
accommodation
Transport
Things to do

Special offers

Popular holiday packages

e.g. Winter sun
Disneyworld
Skiing

Search
results

Country-specific info

Accommodation / attraction reviews

Visa / entry requirements

Money / Currency

Languages

Insurance

Buy
holidays / other
things

4. Next, try to sort all these content items into groups, to give you an idea of what parts might live together on different pages.
This is very similar to a technique called [Card sorting](#).

Search

Flights
Hotels
Other accommodation
Transport
Things to do

Country-specific info

Reviews
Visa / entry requirements
Money / currency
Languages

Specials

Special offers
Popular holidays

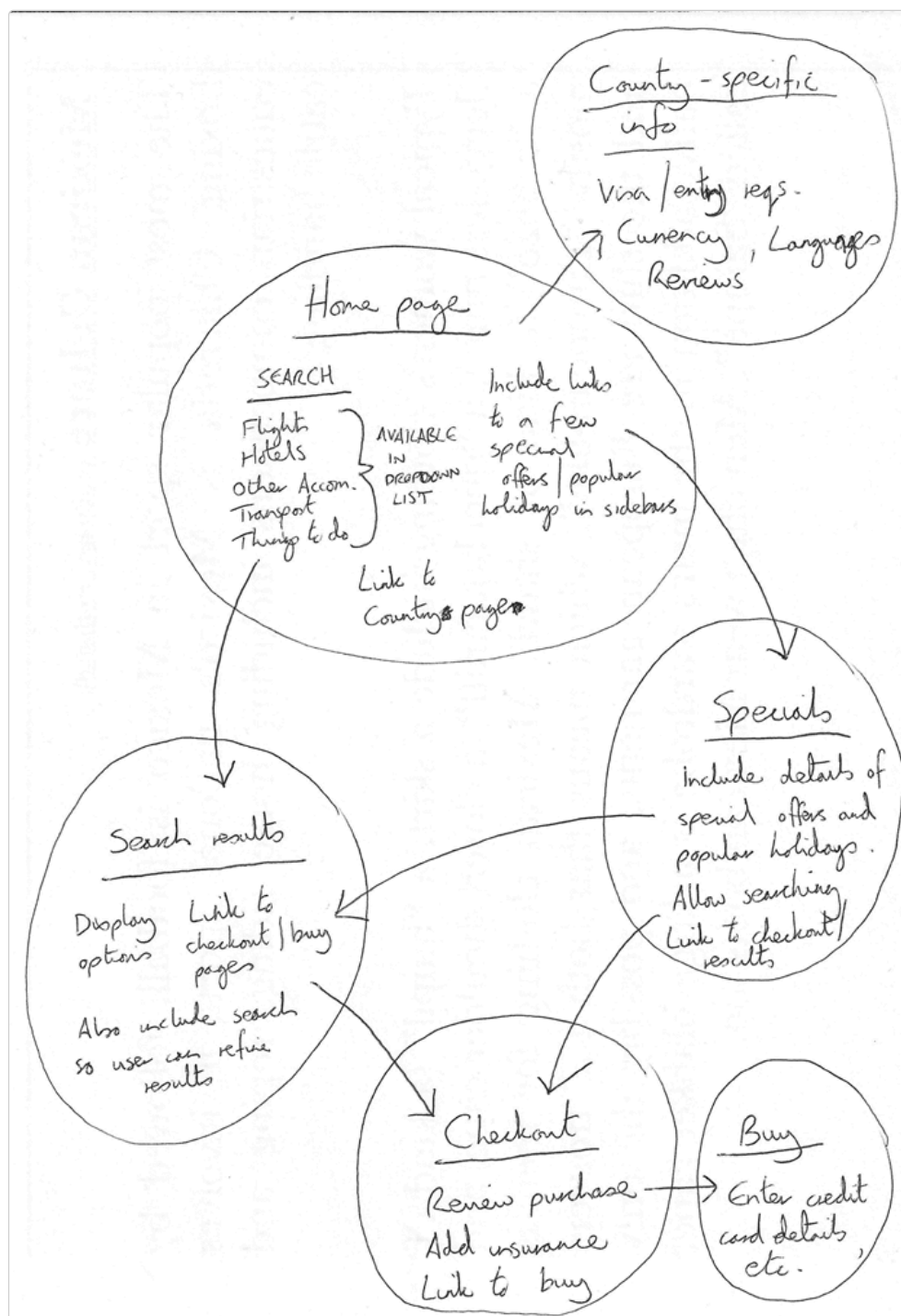
Search results

Also include search
functionality here
to refine results

Buy things

Shopping cart
Checkout
Insurance

5. Now try to sketch a rough sitemap — have a bubble for each page on your site, and draw lines to show the typical workflow between pages. The homepage will probably be in the center, and link to most if not all of the others; most of the pages in a small site should be available from the main navigation, although there are exceptions. You might also want to include notes about how things might be presented.



Active learning: create your own sitemap

Try carrying out the above exercise for a website of your own creation. What would you like to make a site about?

Note: Save your work somewhere; you might need it later on.

Summary

At this point, you should have a better idea about how to structure a web page/site. In the next article of this module, we'll learn how to [debug HTML](#).

Help improve MDN

Was this page helpful to you?



Yes

No

[Learn how to contribute.](#)

This page was last modified on Sep 14, 2023 by [MDN contributors](#).



The HTML5 input types

In the [previous article](#) we looked at the `<input>` element, covering the original values of the `type` attribute available since the early days of HTML. Now we'll look at the functionality of newer form controls in detail, including some new input types, which were added in HTML5 to allow the collection of specific types of data.

Prerequisites:	A basic understanding of HTML .
Objective:	To understand the newer input type values available to create native form controls, and how to implement them using HTML.

Note: Most of the features discussed in this article have wide support across browsers. We'll note any exceptions. If you want more detail on browser support, you should consult our [HTML forms element reference](#), and in particular our extensive [<input> types](#) reference.

Because HTML form control appearance may be quite different from a designer's specifications, web developers sometimes build their own custom form controls. We cover this in an advanced tutorial: [How to build custom form widgets](#).

Email address field

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

```
HTML
<input type="email" id="email" name="email" />
```

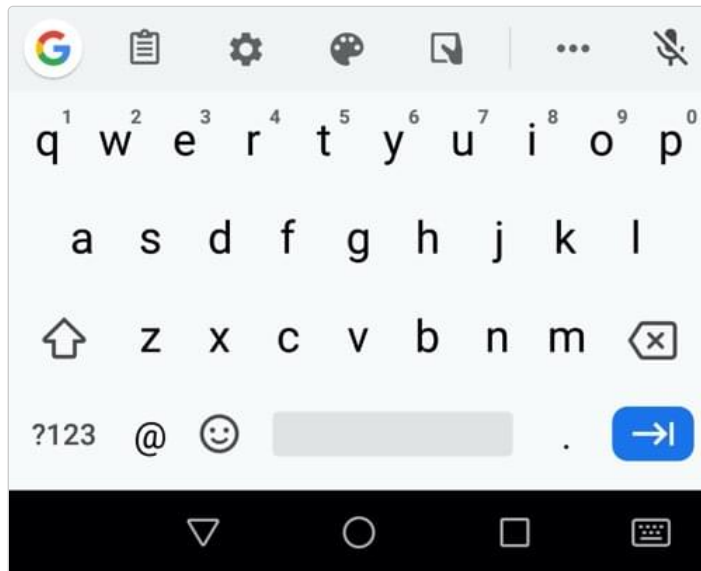
When this `type` is used, the user is required to type a valid email address into the field. Any other content causes the browser to display an error when the form is submitted. You can see this in action in the below screenshot.



You can also use the `multiple` attribute in combination with the `email` input type to allow several email addresses to be entered in the same input (separated by commas):

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

On some devices — notably, touch devices with dynamic keyboards like smartphones — a different virtual keypad might be presented that is more suitable for entering email addresses, including the `@` key. See the Firefox for Android keyboard screenshot below for an example:



Note: You can find examples of the basic text input types at [basic input examples](#) (see the [source code](#) also).

This is another good reason for using these newer input types, improving the user experience for users of these devices.

Client-side validation

As you can see above, `email` — along with other newer `input` types — provides built-in *client-side* error validation, performed by the browser before the data gets sent to the server. It *is* a helpful aid to guide users to fill out a form accurately, and it can save time: it is useful to know that your data is not correct immediately, rather than having to wait for a round trip to the server.

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

Note that `a@b` is a valid email address according to the default provided constraints. This is because the `email` input type allows intranet email addresses by default. To implement different validation behavior, you can use the [pattern](#) attribute, and you can also customize the error messages; we'll talk about how to use these features in the [Client-side form validation](#) article later on.

Note: If the data entered is not an email address, the `:invalid` pseudo-class will match, and the `validityState.typeMismatch` property will return `true`.

Search field

Search fields are intended to be used to create search boxes on pages and apps. This type of field is set by using the value `search` for the `type` attribute:

HTML

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

The main difference between a `text` field and a `search` field is how the browser styles its appearance. Often, `search` fields are rendered with rounded corners; they also sometimes display an "X", which clears the field of any value when clicked. Additionally, on devices with dynamic keyboards, the keyboard's enter key may read **"search"**, or display a magnifying glass icon.

The below screenshots show a non-empty search field in Firefox 71, Safari 13, and Chrome 79 on macOS, and Edge 18 and Chrome 79 on Windows 10. Note that the clear icon only appears if the field has a value, and, apart from Safari, it is only displayed when the field is focused.



Another worth-noting feature is that the values of a search field can be automatically saved and re-used to offer auto-completion

[mdn web docs](#)

Phone number field

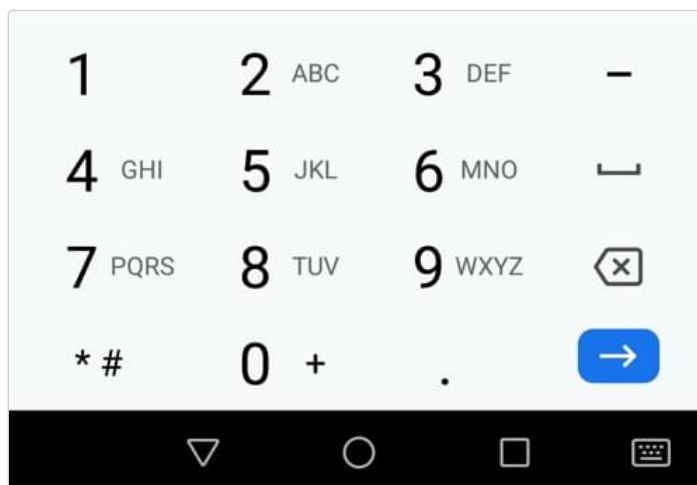
A special field for filling in phone numbers can be created using `tel` as the value of the `type` attribute:

HTML

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

When accessed via a touch device with a dynamic keyboard, most devices will display a numeric keypad when `type="tel"` is encountered, meaning this type is useful whenever a numeric keypad is useful, and doesn't just have to be used for telephone numbers.

The following Firefox for Android keyboard screenshot provides an example:



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 means it may include letters, etc.).

As we mentioned earlier, the `pattern` attribute can be used to enforce constraints, which you'll learn about in [Client-side form validation](#).

URL field

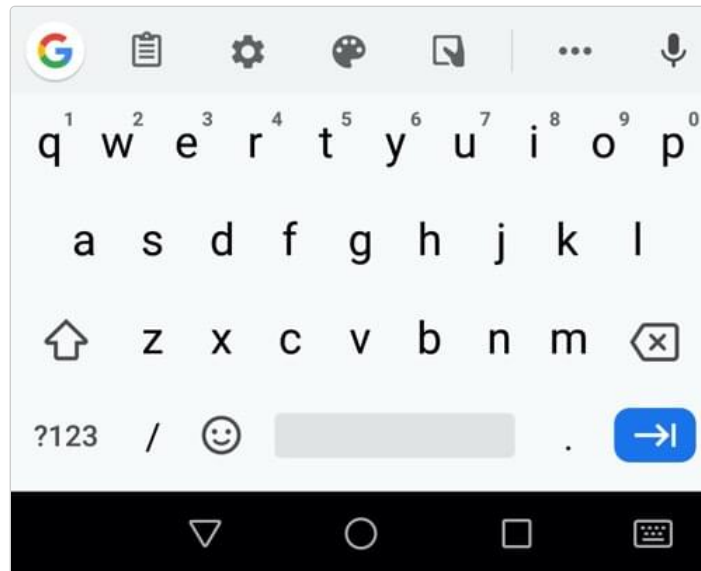
A special type of field for entering URLs can be created using the value `url` for the `type` attribute:

HTML

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

It adds special validation constraints to the field. The browser will report an error if no protocol (such as `http:`) is entered, or if the URL is otherwise malformed. On devices with dynamic keyboards, the default keyboard will often display some or all of the colon, period, and forward slash as default keys.

See below for an example (taken on Firefox for Android):

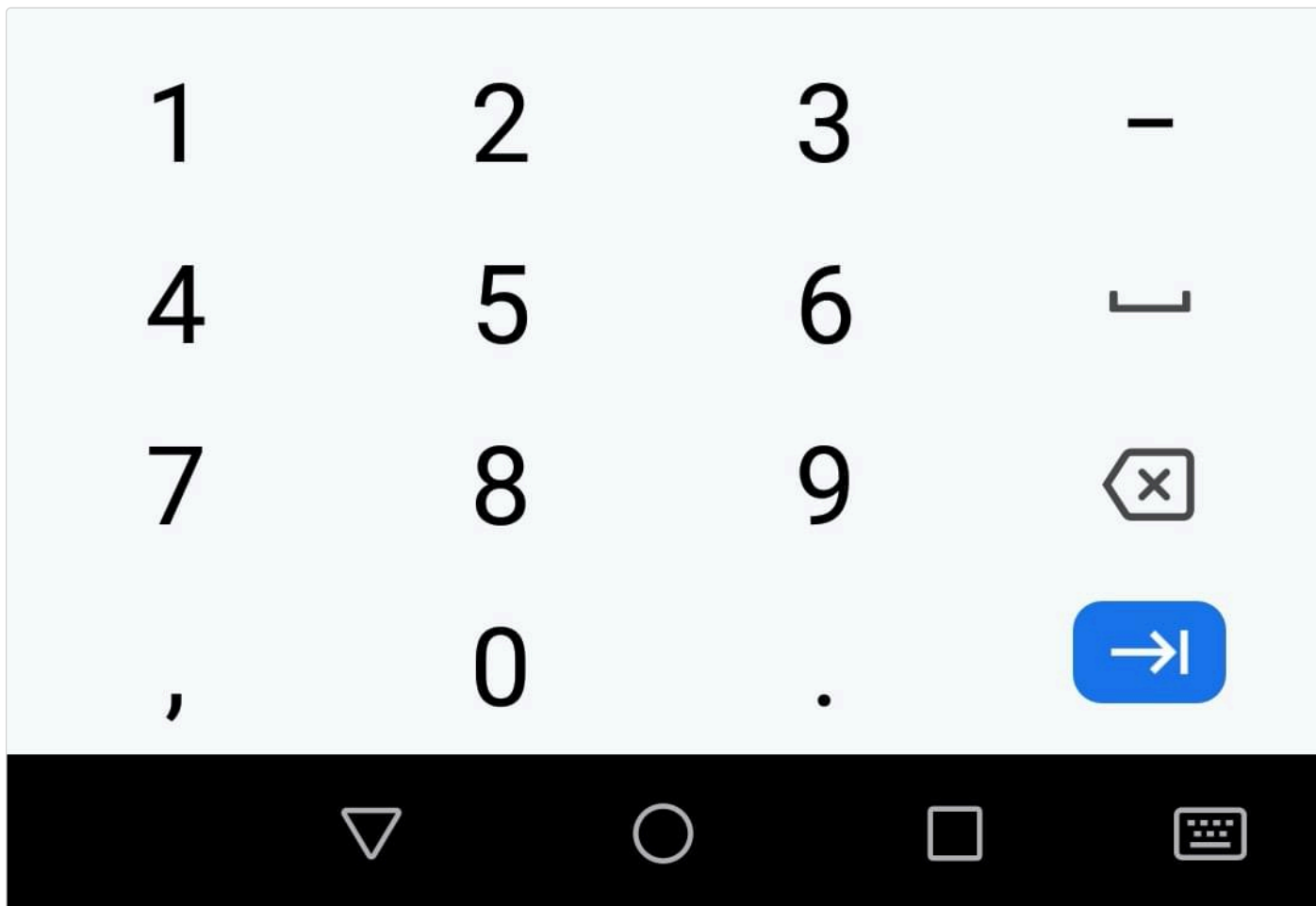


Note: Just because the URL is well-formed doesn't necessarily mean that it refers to a location that actually exists!

Numeric field

Controls for entering numbers can be created with an `<input type="number">`. This control looks like a text field but allows only floating-point numbers, and usually provides buttons in the form of a spinner to increase and decrease the value of the control. On devices with dynamic keyboards, the numeric keyboard is generally displayed.

The following screenshot (from Firefox for Android) provides an example:



With the `number` input type, you can constrain the minimum and maximum values allowed by setting the [min](#) and [max](#) attributes.

You can also use the `step` attribute to set the increment increase and decrease caused by pressing the spinner buttons. By default, the number input type only validates if the number is an integer. To allow float numbers, specify [step="any"](#). If omitted, the `step` value defaults to `1`, meaning only whole numbers are valid.

Let's look at some examples. The first one below creates a number control whose value is restricted to any value between `1` and `10`, and whose increase and decrease buttons change its value by `2`.

HTML

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

The second one creates a number control whose value is restricted to any value between `0` and `1` inclusive, and whose increase and decrease buttons change its value by `0.01`.

HTML


```
<input type="number" name="change" id="pennies" min="0" max="1" step="0.01" />
```

The `number` input type makes sense when the range of valid values is limited, for example a person's age or height. If the range is too large for incremental increases to make sense (such as USA ZIP codes, which range from `00001` to `99999`), the `tel` type might be a better option; it provides the numeric keypad while forgoing the number's spinner UI feature.

Slider controls

Another way to pick a number is to use a **slider**. You see these quite often on sites like house-buying sites where you want to set a maximum property price to filter by. Let's look at a live example to illustrate this:

Choose a maximum house price:



Submit

Usage-wise, sliders are less accurate than text fields. Therefore, they are used to pick a number whose *precise* value is not necessarily important.

A slider is created using the `<input>` with its `type` attribute set to the value `range`. The slider-thumb can be moved via mouse or touch, or with the arrows of the keypad.

It's important to properly configure your slider. To that end, it's highly recommended that you set the `min`, `max`, and `step` attributes which set the minimum, maximum, and increment values, respectively.

Let's look at the code behind the above example, so you can see how it's done. First of all, the basic HTML:

HTML

```
<label for="price">Choose a maximum house price: </label>
<input
  type="range"
  name="price"
  id="price"
  min="50000"
  max="500000"
  step="100"
  value="250000" />
<output class="price-output" for="price"></output>
```

This example creates a slider whose value may range between `50000` and `500000`, which increments/decrements by `100` at a time. We've given it a default value of `250000`, using the `value` attribute.

One problem with sliders is that they don't offer any kind of visual feedback as to what the current value is. This is why we've included an `<output>` element to contain the current value. You could display an input value or the output of a calculation inside any element, but `<output>` is special — like `<label>` — and it can take a `for` attribute that allows you to associate it with the element or elements that the output value came from.

To actually display the current value, and update it as it changed, you must use JavaScript, but this is relatively easy to do:

JS

```
const price = document.querySelector("#price");
const output = document.querySelector(".price-output");

output.textContent = price.value;

price.addEventListener("input", () => {
  output.textContent = price.value;
});
```

Here we store references to the `range` input and the `output` in two variables. Then we immediately set the `output`'s `textContent` to the current `value` of the input. Finally, an event listener is set to ensure that whenever the range slider is moved, the `output`'s

textContent is updated to the new value.

Note: There is a nice tutorial covering this subject on CSS Tricks: [The Output Element](#) .

Date and time pickers

Gathering date and time values has traditionally been a nightmare for web developers. For a good user experience, it is important to provide a calendar selection UI, enabling users to select dates without necessitating context switching to a native calendar application or potentially entering them in differing formats that are hard to parse. The last minute of the previous millennium can be expressed in the following different ways, for example: 1999/12/31, 23:59 or 12/31/99T11:59PM.

HTML date controls are available to handle this specific kind of data, providing calendar widgets and making the data uniform.

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. Here's a live example that falls back to `<select>` elements in non-supporting browsers:

Choose a date and time for your party: ✓

Let's look at the different available types in brief. Note that the usage of these types is quite complex, especially considering browser support (see below); to find out the full details, follow the links below to the reference pages for each type, including detailed examples.

datetime-local

[<input type="datetime-local">](#) creates a widget to display and pick a date with time with no specific time zone information.

HTML

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

month

[<input type="month">](#) creates a widget to display and pick a month with a year.

HTML

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

time

[<input type="time">](#) creates a widget to display and pick a time value. While time may *display* in 12-hour format, the *value returned* is in 24-hour format.

HTML

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

week

`<input type="week">` creates a widget to display and pick a week number and its year.

Weeks start on Monday and run to Sunday. Additionally, the first week 1 of each year contains the first Thursday of that year — which may not include the first day of the year, or may include the last few days of the previous year.

HTML

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

Constraining date/time values

All date and time controls can be constrained using the [min](#) and [max](#) attributes, with further constraining possible via the [step](#) attribute (whose value varies according to input type).

HTML

```
<label for="myDate">When are you available this summer?</label>
<input
  type="date"
  name="myDate"
  min="2013-06-01"
  max="2013-08-31"
  step="7"
  id="myDate" />
```

Color picker control

Colors are always a bit difficult to handle. There are many ways to express them: RGB values (decimal or hexadecimal), HSL values, keywords, and so on.

A `color` control can be created using the `<input>` element with its `type` attribute set to the value `color` :

HTML

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

Clicking a color control generally displays the operating system's default color-picking functionality for you to choose.

Here is a live example for you to try out:

Pick a color:

Submit

The value returned is always a lowercase 6-value hexadecimal color.

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see [Test your skills: HTML5 controls](#).

Summary

That brings us to the end of our tour of the HTML5 form input types. There are a few other control types that cannot be easily grouped together due to their very specific behaviors, but which are still essential to know about. We cover those in the next article.

Advanced Topics

- [How to build custom form controls](#)
- [Sending forms through JavaScript](#)
- [Property compatibility table for form widgets](#)

Help improve MDN

Was this page helpful to you?

[Learn how to contribute.](#)

This page was last modified on Jan 1, 2024 by [MDN contributors](#).

