# Getting started with HTML

* [Overview: Introduction to HTML](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML)
* [Next](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/The_head_metadata_in_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 computer literacy, [basic software installed](https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/Installing_basic_software), and basic knowledge of [working with files](https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/Dealing_with_files). |
| **Objective:** | To gain basic familiarity with HTML, and practice writing a few HTML elements. |

## [What is HTML?](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#what_is_html)

[HTML](https://developer.mozilla.org/en-US/docs/Glossary/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](https://developer.mozilla.org/en-US/docs/Glossary/Element), 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](https://developer.mozilla.org/en-US/docs/Glossary/Tag) 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>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/p)) element:

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

Copy to Clipboard

**Note:** Tags in HTML are not case-sensitive. This means they can be written in uppercase or lowercase. For example, a [<title>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/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](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#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](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#active_learning_creating_your_first_html_element)

Edit the line below in the "Editable code" area by wrapping it with the tags <em> and </em>. To open the element, put the opening tag <em> at the start of the line. To close the element, put the closing tag </em> 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.

### [Nesting elements](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#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 [<strong>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/strong) element, which means that the word is to have strong(er) text formatting:

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

Copy to Clipboard

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:

<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.

### [Block versus inline elements](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#block_versus_inline_elements)

There are two important categories of elements to know in HTML: block-level elements and inline elements.

* Block-level elements form a visible block on a page. A block-level element appears on a new line following the content that precedes it. Any content that follows a block-level element also appears on a new line. Block-level elements are usually structural elements on the page. For example, a block-level element might represent headings, paragraphs, lists, navigation menus, or footers. A block-level element wouldn't be nested inside an inline element, but it might be nested inside another block-level element.
* Inline elements are contained within block-level elements, and surround only small parts of the document's content (not entire paragraphs or groupings of content). An inline element will not cause a new line to appear in the document. It is typically used with text, for example an [<a>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/a) element creates a hyperlink, and elements such as [<em>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/em) or [<strong>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/strong) create emphasis.

Consider the following example:

<em>first</em><em>second</em><em>third</em>

<p>fourth</p>

<p>fifth</p>

<p>sixth</p>

Copy to Clipboard

[<em>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/em) is an inline element. As you see below, the first three elements sit on the same line, with no space in between. On the other hand, [<p>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/p) is a block-level element. Each p element appears on a new line, with space above and below. (The spacing is due to default [CSS styling](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps) that the browser applies to paragraphs.)

**Note:** HTML5 redefined the element categories: see [Element content categories](https://html.spec.whatwg.org/multipage/indices.html#element-content-categories). While these definitions are more accurate and less ambiguous than their predecessors, the new definitions are a lot more complicated to understand than block and inline. This article will stay with these two terms.

**Note:** The terms block and inline, as used in this article, should not be confused with [the types of CSS boxes](https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/The_box_model#block_and_inline_boxes) that have the same names. While the names correlate by default, changing the CSS display type doesn't change the category of the element, and doesn't affect which elements it can contain and which elements it can be contained in. One reason HTML5 dropped these terms was to prevent this rather common confusion.

**Note:** Find useful reference pages that include lists of block and inline elements. See [Block-level elements](https://developer.mozilla.org/en-US/docs/Web/HTML/Block-level_elements) and [Inline elements](https://developer.mozilla.org/en-US/docs/Web/HTML/Inline_elements).

### [Void elements](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#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](https://developer.mozilla.org/en-US/docs/Glossary/Void_element). For example, the [<img>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/img) element embeds an image file onto a page:

<img

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

Copy to Clipboard

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: <img src="images/cat.jpg" alt="cat" />. However, it is also a valid syntax, and you may do this when you want your HTML to be valid XML.

## [Attributes](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#attributes)

Elements can also have attributes. Attributes look like this:



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](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#active_learning_adding_attributes_to_an_element)

Another example of an element is [<a>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/a). This stands for anchor. An anchor can make the text it encloses into a hyperlink. Anchors can take a number of attributes, but several are as follows:

href

This attribute's value specifies the web address for the link. For example: href="https://www.mozilla.org/".

title

The title attribute specifies extra information about the link, such as a description of the page that is being linked to. For example, title="The Mozilla homepage". This appears as a tooltip when a cursor hovers over the element.

target

The target attribute specifies the browsing context used to display the link. For example, target="\_blank" will display the link in a new tab. If you want to display the linked content in the current tab, just omit this attribute.

Edit the line below in the Input area to turn it into a link to your favorite website.

1. Add the <a> element.
2. Add the href attribute and the title attribute.
3. Specify the target attribute to open the link in the new tab.

You will be able to see your changes update live in the Output area. You should see a link—that when hovered over—displays the value of the title attribute and, when clicked, opens a new tab and navigates to the web address in the href attribute. Remember that you need to include a space between the element name, and between each attribute.

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.

### [Boolean attributes](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#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](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#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:

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

Copy to Clipboard

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

<!-- 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" />

Copy to Clipboard

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

### [Omitting quotes around attribute values](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#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. For example, if we revisit our link example from earlier, we could write a basic version with only the href attribute, like this:

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

Copy to Clipboard

However, as soon as we add the title attribute in this way, there are problems:

<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!

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

### [Single or double quotes?](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#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:

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

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

Copy to Clipboard

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

<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:

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

Copy to Clipboard

To use quote marks inside other quote marks of the same type (single quote or double quote), use [HTML entities](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#entity_references_including_special_characters_in_html). For example, this will break:

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

Instead, you need to do this:

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

Copy to Clipboard

## [Anatomy of an HTML document](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#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:

<!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>

Copy to Clipboard

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

Copy to Clipboard

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!

1. <html></html>: The [<html>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/html) element. This element wraps all the content on the page. It is sometimes known as the root element.
2. <head></head>: The [<head>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/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.
3. <meta charset="utf-8">: The [<meta>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/meta) element. This element represents metadata that cannot be represented by other HTML meta-related elements, like [<base>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/base), [<link>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/link), [<script>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script), [<style>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/style) or [<title>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/title). The [charset](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/meta#charset) attributes sets the character set for your document to 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.
4. <title></title>: The [<title>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/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.
5. <body></body>: The [<body>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/body) element. This contains all the content that displays on the page, including text, images, videos, games, playable audio tracks, or whatever else.

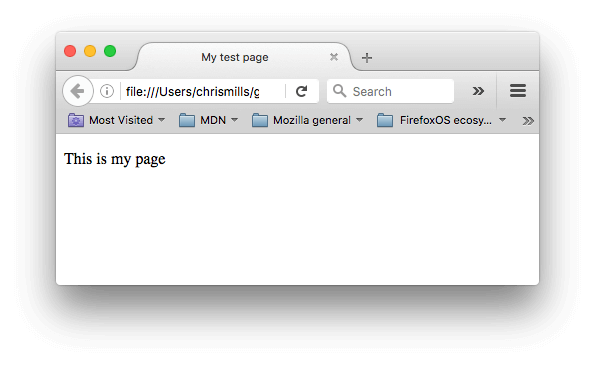
### [Active learning: Adding some features to an HTML document](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#active_learning_adding_some_features_to_an_html_document)

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](https://github.com/mdn/learning-area/blob/main/html/introduction-to-html/getting-started/index.html).

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>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/body) element, in this case). Sharpen your skills by implementing the following tasks:

* Just below the opening tag of the [<body>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/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 <strong> opening tag and </strong> closing tag.
* Add a link to your paragraph, as [explained earlier in the article](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#active_learning_adding_attributes_to_an_element).
* Add an image to your document. Place it below the paragraph, as [explained earlier in the article](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#empty_elements). 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.

### [Whitespace in HTML](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#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:

<p>Dogs are silly.</p>

<p>Dogs are

silly.</p>

Copy to Clipboard

No matter how much whitespace you use inside HTML element content (which can include one or more space character, 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.

## [Entity references: Including special characters in HTML](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#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 (;).

| **Literal character** | **Character reference equivalent** |
| --- | --- |
| < | &lt; |
| > | &gt; |
| " | &quot; |
| ' | &apos; |
| & | &amp; |

The character reference equivalent could be easily remembered because the text it uses can be seen as less than for '&lt;', quotation for ' &quot; ' and similarly for others. To find more about entity reference, see [List of XML and HTML character entity references](https://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references) (Wikipedia).

In the example below, there are two paragraphs:

<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>

Copy to Clipboard

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.

**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](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/The_head_metadata_in_HTML#specifying_your_documents_character_encoding).

## [HTML comments](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#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:

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

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

Copy to Clipboard

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

# Document and website structure

* [Previous](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Advanced_text_formatting)
* [Overview: Introduction to HTML](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML)
* [Next](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Debugging_HTML)

In addition to defining individual parts of your page (such as "a paragraph" or "an image"), [HTML](https://developer.mozilla.org/en-US/docs/Glossary/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](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started). HTML text formatting, as covered in [HTML text fundamentals](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/HTML_text_fundamentals). How hyperlinks work, as covered in [Creating hyperlinks](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/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](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Document_and_website_structure#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](https://developer.mozilla.org/en-US/docs/Learn/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!

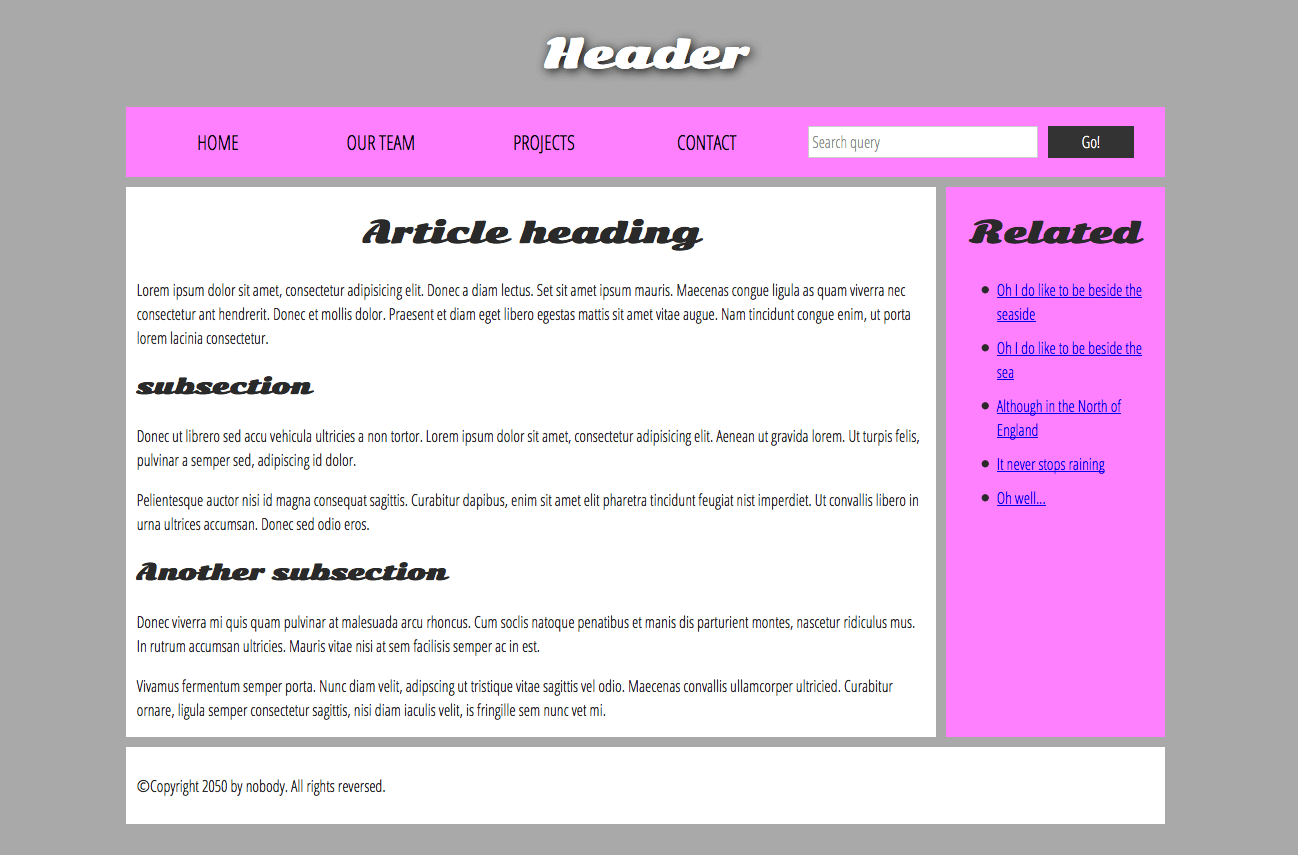
sidebar:

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](https://developer.mozilla.org/en-US/docs/Glossary/SEO) purposes, by providing links for quick access to popular content.

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



**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](https://developer.mozilla.org/en-US/docs/Learn/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](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps) module to start learning how to style your site.

## [HTML for structuring content](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Document_and_website_structure#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](https://www.color-blindness.com/) 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](https://en.wikipedia.org/wiki/Visual_impairment), while the total population was [around 7.5 billion](https://en.wikipedia.org/wiki/World_human_population#/media/File:World_population_history.svg)).

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](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/HTML_text_fundamentals#why_do_we_need_structure).

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

* **header:** [<header>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/header).
* **navigation bar:** [<nav>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/nav).
* **main content:** [<main>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/main), with various content subsections represented by [<article>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/article), [<section>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/section), and [<div>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div) elements.
* **sidebar:** [<aside>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/aside); often placed inside [<main>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/main).
* **footer:** [<footer>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/footer).

### [Active learning: exploring the code for our example](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Document_and_website_structure#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](https://github.com/mdn/learning-area/blob/main/html/introduction-to-html/document_and_website_structure/index.html)). 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.

<!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 librero sed accu vehicula ultricies a non tortor. Lorem ipsum

dolor sit amet, consectetur adipisicing 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

soclis 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, adipscing ut

tristique vitae sagittis vel odio. Maecenas convallis ullamcorper

ultricied. 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>

Copy to Clipboard

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](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Document_and_website_structure#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](https://developer.mozilla.org/en-US/docs/Web/HTML/Element). For now, these are the main definitions that you should try to understand:

* [<main>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/main) is for content unique to this page. Use <main> only once per page, and put it directly inside [<body>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/body). Ideally this shouldn't be nested within other elements.
* [<article>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/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>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/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](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/HTML_text_fundamentals); 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>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/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>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/header) represents a group of introductory content. If it is a child of [<body>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/body) it defines the global header of a webpage, but if it's a child of an [<article>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/article) or [<section>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/section) it defines a specific header for that section (try not to confuse this with [titles and headings](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/The_head_metadata_in_HTML#adding_a_title)).
* [<nav>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/nav) contains the main navigation functionality for the page. Secondary links, etc., would not go in the navigation.
* [<footer>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/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](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Document_and_website_structure#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](https://developer.mozilla.org/en-US/docs/Glossary/CSS) or [JavaScript](https://developer.mozilla.org/en-US/docs/Glossary/JavaScript). For cases like these, HTML provides the [<div>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div) and [<span>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/span) elements. You should use these preferably with a suitable [class](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes#class) attribute, to provide some kind of label for them so they can be easily targeted.

[<span>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/span) 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:

<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>

Copy to Clipboard

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>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/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>

<img src="../products/3333-0985/thumb.png" alt="Silver earrings" />

</li>

<li>…</li>

</ul>

<p>Total cost: $237.89</p>

</div>

Copy to Clipboard

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:** Divs 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](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Document_and_website_structure#line_breaks_and_horizontal_rules)

Two elements that you'll use occasionally and will want to know about are [<br>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/br) and [<hr>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/hr).

#### <br>: the line break element

<br> 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:

<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>

Copy to Clipboard

Without the <br> elements, the paragraph would just be rendered in one long line (as we said earlier in the course, [HTML ignores most whitespace](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Getting_started#whitespace_in_html)); with <br> elements in the code, the markup renders like this:

#### <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.

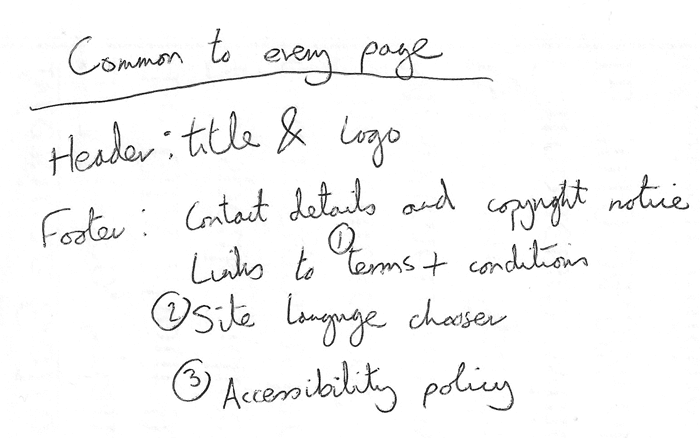
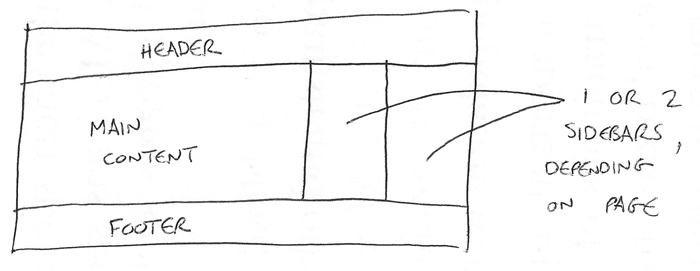
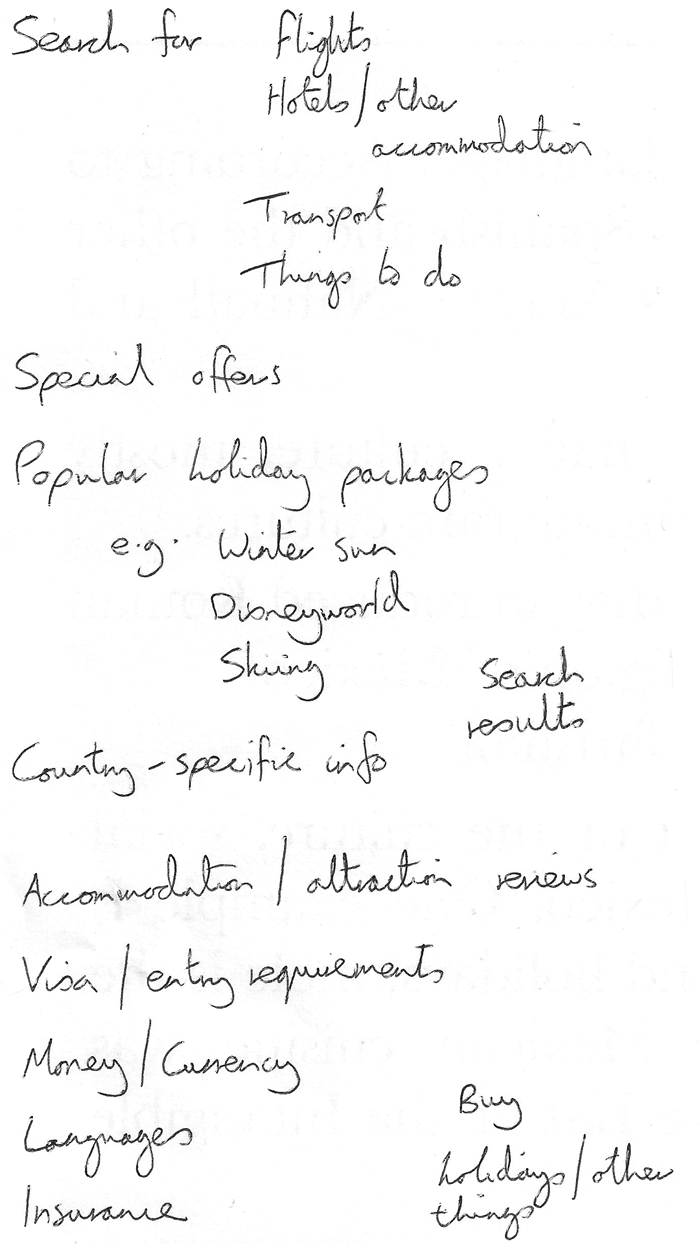
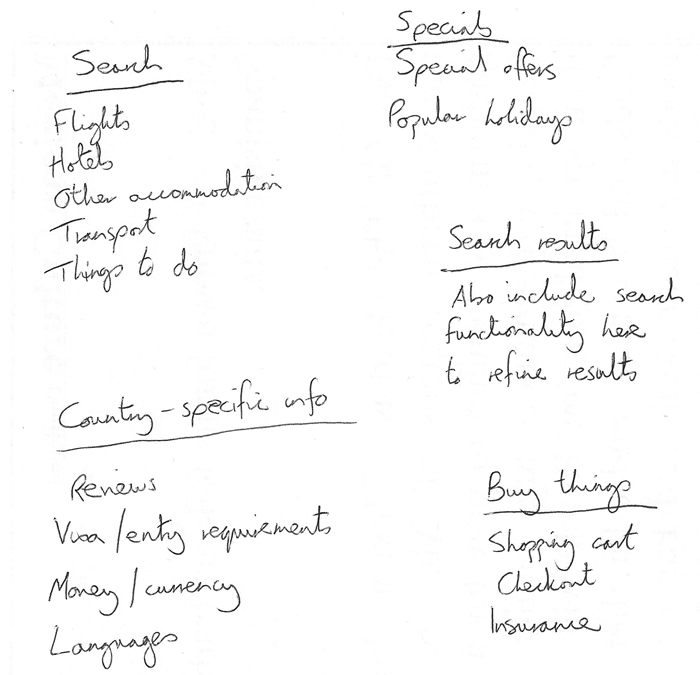
</p>

Copy to Clipboard

Would render like this:

## [Planning a simple website](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Document_and_website_structure#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](https://developer.mozilla.org/en-US/docs/Glossary/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.
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](https://developer.mozilla.org/en-US/docs/Glossary/Card_sorting).
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.

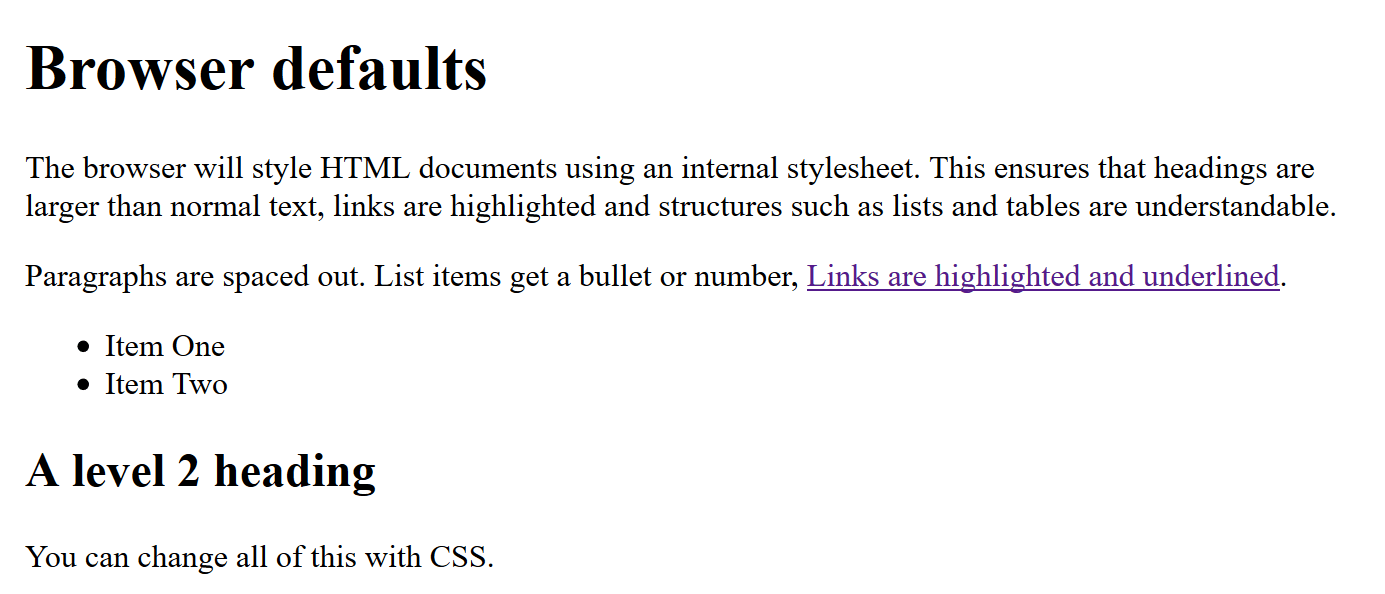
# What is CSS?

* [Overview: First steps](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps)
* [Next](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/Getting_started)

[**CSS**](https://developer.mozilla.org/en-US/docs/Glossary/CSS) (Cascading Style Sheets) allows you to create great-looking web pages, but how does it work under the hood? This article explains what CSS is with a simple syntax example and also covers some key terms about the language.

|  |  |
| --- | --- |
| **Prerequisites:** | Basic computer literacy, [basic software installed](https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/Installing_basic_software), basic knowledge of [working with files](https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/Dealing_with_files), and HTML basics (study [Introduction to HTML](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML).) |
| **Objective:** | To learn what CSS is. |

In the [Introduction to HTML](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML) module, we covered what HTML is and how it is used to mark up documents. These documents will be readable in a web browser. Headings will look larger than regular text, paragraphs break onto a new line and have space between them. Links are colored and underlined to distinguish them from the rest of the text. What you are seeing are the browser's default styles — very basic styles — that the browser applies to HTML to make sure that the page will be basically readable even if no explicit styling is specified by the author of the page.



However, the web would be a boring place if all websites looked like that. Using CSS, you can control exactly how HTML elements look in the browser, presenting your markup using whatever design you like.

For more on browser/default styles, check out the following video:

## [What is CSS for?](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS#what_is_css_for)

As we have mentioned before, CSS is a language for specifying how documents are presented to users — how they are styled, laid out, etc.

A **document** is usually a text file structured using a markup language — [HTML](https://developer.mozilla.org/en-US/docs/Glossary/HTML) is the most common markup language, but you may also come across other markup languages such as [SVG](https://developer.mozilla.org/en-US/docs/Glossary/SVG) or [XML](https://developer.mozilla.org/en-US/docs/Glossary/XML).

**Presenting** a document to a user means converting it into a form usable by your audience. [Browsers](https://developer.mozilla.org/en-US/docs/Glossary/Browser), like [Firefox](https://developer.mozilla.org/en-US/docs/Glossary/Mozilla_Firefox), [Chrome](https://developer.mozilla.org/en-US/docs/Glossary/Google_Chrome), or [Edge](https://developer.mozilla.org/en-US/docs/Glossary/Microsoft_Edge), are designed to present documents visually, for example, on a computer screen, projector, or printer.

**Note:** A browser is sometimes called a [user agent](https://developer.mozilla.org/en-US/docs/Glossary/User_agent), which basically means a computer program that represents a person inside a computer system. Browsers are the main type of user agents we think of when talking about CSS, however, they are not the only ones. There are other user agents available, such as those that convert HTML and CSS documents into PDFs to be printed.

CSS can be used for very basic document text styling — for example, for changing the [color](https://developer.mozilla.org/en-US/docs/Web/CSS/color_value) and [size](https://developer.mozilla.org/en-US/docs/Web/CSS/font-size) of headings and links. It can be used to create a layout — for example, [turning a single column of text into a layout](https://developer.mozilla.org/en-US/docs/Web/CSS/Layout_cookbook/Column_layouts) with a main content area and a sidebar for related information. It can even be used for effects such as [animation](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Animations). Have a look at the links in this paragraph for specific examples.

## [CSS syntax](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS#css_syntax)

CSS is a rule-based language — you define the rules by specifying groups of styles that should be applied to particular elements or groups of elements on your web page.

For example, you can decide to have the main heading on your page to be shown as large red text. The following code shows a very simple CSS rule that would achieve the styling described above:

h1 {

color: red;

font-size: 5em;

}

Copy to Clipboard

* In the above example, the CSS rule opens with a [selector](https://developer.mozilla.org/en-US/docs/Glossary/CSS_Selector). This selects the HTML element that we are going to style. In this case, we are styling level one headings ([h1](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Heading_Elements)).
* We then have a set of curly braces { }.
* Inside the braces will be one or more **declarations**, which take the form of **property** and **value** pairs. We specify the property (color in the above example) before the colon, and we specify the value of the property after the colon (red in this example).
* This example contains two declarations, one for color and the other for font-size. Each pair specifies a property of the element(s) we are selecting ([h1](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Heading_Elements) in this case), then a value that we'd like to give the property.

CSS [properties](https://developer.mozilla.org/en-US/docs/Glossary/Property/CSS) have different allowable values, depending on which property is being specified. In our example, we have the color property, which can take various [color values](https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Values_and_units#color). We also have the font-size property. This property can take various [size units](https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Values_and_units#numbers_lengths_and_percentages) as a value.

A CSS stylesheet will contain many such rules, written one after the other.

h1 {

color: red;

font-size: 5em;

}

p {

color: black;

}

Copy to Clipboard

You will find that you quickly learn some values, whereas others you will need to look up. The individual property pages on MDN give you a quick way to look up properties and their values when you forget or when you want to know what else you can use as a value.

**Note:** You can find links to all the CSS property pages (along with other CSS features) listed on the MDN [CSS reference](https://developer.mozilla.org/en-US/docs/Web/CSS/Reference). Alternatively, you should get used to searching for "mdn css-feature-name" in your favorite search engine whenever you need to find out more information about a CSS feature. For example, try searching for "mdn color" and "mdn font-size"!

## [CSS modules](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS#css_modules)

As there are so many things that you could style using CSS, the language is broken down into modules. You'll see reference to these modules as you explore MDN. Many of the documentation pages are organized around a particular module. For example, you could take a look at the MDN reference to the [Backgrounds and Borders](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Backgrounds_and_Borders) module to find out what its purpose is and the properties and features it contains. In that module, you will also find a link to Specifications that defines the technology (also see the section below).

At this stage, you don't need to worry too much about how CSS is structured; however, it can make it easier to find information if, for example, you are aware that a certain property is likely to be found among other similar things, and is therefore, probably in the same specification.

For a specific example, let's go back to the Backgrounds and Borders module — you might think that it makes logical sense for the [background-color](https://developer.mozilla.org/en-US/docs/Web/CSS/background-color) and [border-color](https://developer.mozilla.org/en-US/docs/Web/CSS/border-color) properties to be defined in this module. And you'd be right.

## [CSS specifications](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS#css_specifications)

All web standards technologies (HTML, CSS, JavaScript, etc.) are defined in giant documents called specifications (or "specs"), which are published by standards organizations (such as the [W3C](https://developer.mozilla.org/en-US/docs/Glossary/W3C), [WHATWG](https://developer.mozilla.org/en-US/docs/Glossary/WHATWG), [ECMA](https://developer.mozilla.org/en-US/docs/Glossary/ECMA), or [Khronos](https://developer.mozilla.org/en-US/docs/Glossary/Khronos)) and define precisely how those technologies are supposed to behave.

CSS is no different — it is developed by a group within the W3C called the [CSS Working Group](https://www.w3.org/Style/CSS/). This group is made of representatives of browser vendors and other companies who have an interest in CSS. There are also other people, known as invited experts, who act as independent voices; they are not linked to a member organization.

New CSS features are developed or specified by the CSS Working Group — sometimes because a particular browser is interested in having some capability, other times because web designers and developers are asking for a feature, and sometimes because the Working Group itself has identified a requirement. CSS is constantly developing, with new features becoming available. However, a key thing about CSS is that everyone works very hard to never change things in a way that would break old websites. A website built in 2000, using the limited CSS available then, should still be usable in a browser today!

As a newcomer to CSS, it is likely that you will find the CSS specs overwhelming — they are intended for engineers to use to implement support for the features in user agents, not for web developers to read to understand CSS. Many experienced developers would much rather refer to MDN documentation or other tutorials. Nevertheless, it is worth knowing that these specs exist and understanding the relationship between the CSS you are using, the browser support (see below), and the specs.

## [Browser support information](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS#browser_support_information)

After a CSS feature has been specified, then it is only useful for us in developing web pages if one or more browsers have implemented the feature. This means that the code has been written to turn the instruction in our CSS file into something that can be output to the screen. We'll look at this process more in the lesson [How CSS works](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works). It is unusual for all browsers to implement a feature at the same time, and so there is usually a gap where you can use some part of CSS in some browsers and not in others. For this reason, being able to check implementation status is useful.

The browser support status is shown on every MDN CSS property page in a table named "Browser compatibility". Consult the information in that table to check if the property can be used on your website. For an example, see the [browser compatibility table for the CSS font-family property](https://developer.mozilla.org/en-US/docs/Web/CSS/font-family#browser_compatibility).

Based on your requirements, you can use the browser compatibility table to check how this property is supported across various browsers, or check if your specific browser and the version you have support the property, or if there are any caveats you should be aware of for the browser and version you are using.

[**How does CSS actually work?**](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works#how_does_css_actually_work)

When a browser displays a document, it must combine the document's content with its style information. It processes the document in a number of stages, which we've listed below. Bear in mind that this is a very simplified version of what happens when a browser loads a webpage, and that different browsers will handle the process in different ways. But this is roughly what happens.

1. The browser loads the HTML (e.g. receives it from the network).
2. It converts the [HTML](https://developer.mozilla.org/en-US/docs/Glossary/HTML) into a [DOM](https://developer.mozilla.org/en-US/docs/Glossary/DOM) (*Document Object Model*). The DOM represents the document in the computer's memory. The DOM is explained in a bit more detail in the next section.
3. The browser then fetches most of the resources that are linked to by the HTML document, such as embedded images, videos, and even linked CSS! JavaScript is handled a bit later on in the process, and we won't talk about it here to keep things simpler.
4. The browser parses the fetched CSS, and sorts the different rules by their selector types into different "buckets", e.g. element, class, ID, and so on. Based on the selectors it finds, it works out which rules should be applied to which nodes in the DOM, and attaches style to them as required (this intermediate step is called a render tree).
5. The render tree is laid out in the structure it should appear in after the rules have been applied to it.
6. The visual display of the page is shown on the screen (this stage is called painting).

[**About the DOM**](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works#about_the_dom)

A DOM has a tree-like structure. Each element, attribute, and piece of text in the markup language becomes a [DOM node](https://developer.mozilla.org/en-US/docs/Glossary/Node/DOM) in the tree structure. The nodes are defined by their relationship to other DOM nodes. Some elements are parents of child nodes, and child nodes have siblings.

Understanding the DOM helps you design, debug and maintain your CSS because the DOM is where your CSS and the document's content meet up. When you start working with browser DevTools you will be navigating the DOM as you select items in order to see which rules apply.

[**A real DOM representation**](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works#a_real_dom_representation)

Rather than a long, boring explanation, let's look at an example to see how a real HTML snippet is converted into a DOM.

Take the following HTML code:

<p>

Let's use:

<span>Cascading</span>

<span>Style</span>

<span>Sheets</span>

</p>

Copy to Clipboard

In the DOM, the node corresponding to our <p> element is a parent. Its children are a text node and the three nodes corresponding to our <span> elements. The SPAN nodes are also parents, with text nodes as their children:

P

├─ "Let's use:"

├─ SPAN

| └─ "Cascading"

├─ SPAN

| └─ "Style"

└─ SPAN

└─ "Sheets"

This is how a browser interprets the previous HTML snippet — it renders the above DOM tree and then outputs it in the browser like so:

[**Applying CSS to the DOM**](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works#applying_css_to_the_dom)

Let's say we add some CSS to our document, to style it. Again, the HTML is as follows:

<p>

Let's use:

<span>Cascading</span>

<span>Style</span>

<span>Sheets</span>

</p>

Copy to Clipboard

Let's suppose we apply the following CSS to it:

span {

border: 1px solid black;

background-color: lime;

}

Copy to Clipboard

The browser parses the HTML and creates a DOM from it. Next, it parses the CSS. Since the only rule available in the CSS has a span selector, the browser sorts the CSS very quickly! It applies that rule to each one of the three <span>s, then paints the final visual representation to the screen.

The updated output is as follows:

In our [Debugging CSS](https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Debugging_CSS) article in the next module we will be using browser DevTools to debug CSS problems, and will learn more about how the browser interprets CSS.

[**What happens if a browser encounters CSS it doesn't understand?**](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works#what_happens_if_a_browser_encounters_css_it_doesnt_understand)

The ["Browser support information" section in the "What is CSS" article](https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS#browser_support_information) mentioned that browsers do not necessarily implement new CSS features at the same time. In addition, many people are not using the latest version of a browser. Given that CSS is being developed all the time, and is therefore ahead of what browsers can recognize, you might wonder what happens if a browser encounters a CSS selector or declaration it doesn't recognize.

The answer is that it does nothing, and just moves on to the next bit of CSS!

If a browser is parsing your rules, and encounters a property or value that it doesn't understand, it ignores it and moves on to the next declaration. It will do this if you have made an error and misspelled a property or value, or if the property or value is just too new and the browser doesn't yet support it.

Similarly, if a browser encounters a selector that it doesn't understand, it will just ignore the whole rule and move on to the next one.

In the example below I have used the British English spelling for color, which makes that property invalid as it is not recognized. So my paragraph has not been colored blue. All of the other CSS have been applied however; only the invalid line is ignored.

<p>I want this text to be large, bold and blue.</p>

Copy to Clipboard

p {

font-weight: bold;

colour: blue; /\* incorrect spelling of the color property \*/

font-size: 200%;

}

Copy to Clipboard

This behavior is very useful. It means that you can use new CSS as an enhancement, knowing that no error will occur if it is not understood — the browser will either get the new feature or not. This enables basic fallback styling.

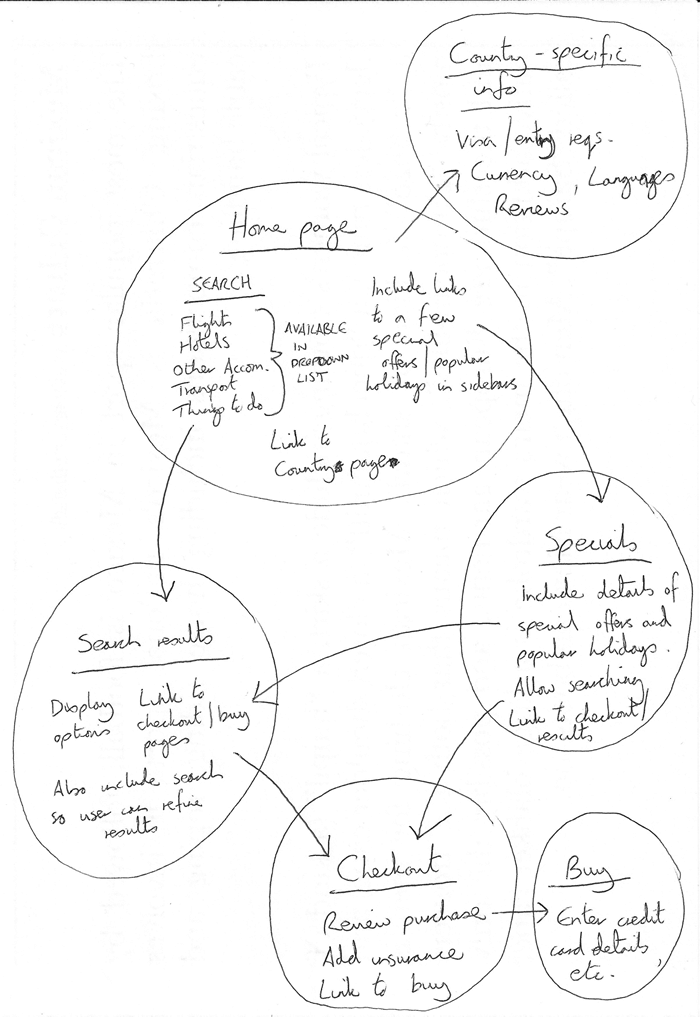
This works particularly well when you want to use a value that is quite new and not supported everywhere. For example, some older browsers do not support calc() as a value. I might give a fallback width for a box in pixels, then go on to give a width with a calc() value of 100% - 50px. Old browsers will use the pixel version, ignoring the line about calc() as they don't understand it. New browsers will interpret the line using pixels, but then override it with the line using calc() as that line appears later in the cascade.

.box {

width: 500px;

width: calc(100% - 50px);

}

1. 

### [Active learning: create your own sitemap](https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Document_and_website_structure#active_learning_create_your_own_sitemap)

# Getting started with React

* [Previous](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Main_features)
* [Overview: Client-side JavaScript frameworks](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks)
* [Next](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_todo_list_beginning)

In this article we will say hello to React. We'll discover a little bit of detail about its background and use cases, set up a basic React toolchain on our local computer, and create and play with a simple starter app — learning a bit about how React works in the process.

|  |  |
| --- | --- |
| **Prerequisites:** | Familiarity with the core [HTML](https://developer.mozilla.org/en-US/docs/Learn/HTML), [CSS](https://developer.mozilla.org/en-US/docs/Learn/CSS), and [JavaScript](https://developer.mozilla.org/en-US/docs/Learn/JavaScript) languages, knowledge of the [terminal/command line](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Command_line).  React uses an HTML-in-JavaScript syntax called JSX (JavaScript and XML). Familiarity with both HTML and JavaScript will help you to learn JSX, and better identify whether bugs in your application are related to JavaScript or to the more specific domain of React. |
| **Objective:** | To set up a local React development environment, create a start app, and understand the basics of how it works. |

## [Hello React](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#hello_react)

As its official tagline states, [React](https://reactjs.org/) is a library for building user interfaces. React is not a framework – it's not even exclusive to the web. It's used with other libraries to render to certain environments. For instance, [React Native](https://reactnative.dev/) can be used to build mobile applications.

To build for the web, developers use React in tandem with [ReactDOM](https://reactjs.org/docs/react-dom.html" \t "_blank). React and ReactDOM are often discussed in the same spaces as — and utilized to solve the same problems as — other true web development frameworks. When we refer to React as a "framework", we're working with that colloquial understanding.

React's primary goal is to minimize the bugs that occur when developers are building UIs. It does this through the use of components — self-contained, logical pieces of code that describe a portion of the user interface. These components can be composed together to create a full UI, and React abstracts away much of the rendering work, leaving you to concentrate on the UI design.

## [Use cases](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#use_cases)

Unlike the other frameworks covered in this module, React does not enforce strict rules around code conventions or file organization. This allows teams to set conventions that work best for them, and to adopt React in any way they would like to. React can handle a single button, a few pieces of an interface, or an app's entire user interface.

While React can be used for [small pieces of an interface](https://reactjs.org/docs/add-react-to-a-website.html), it's not as easy to "drop into" an application as a library like jQuery, or even a framework like Vue — it is more approachable when you build your entire app with React.

In addition, many of the developer-experience benefits of a React app, such as writing interfaces with JSX, require a compilation process. Adding a compiler like Babel to a website makes the code on it run slowly, so developers often set up such tooling with a build step. React arguably has a heavy tooling requirement, but it can be learned.

This article is going to focus on the use case of using React to render the entire user interface of an application, using tooling provided by Facebook's own [create-react-app](https://create-react-app.dev/) tool.

## [How does React use JavaScript?](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#how_does_react_use_javascript)

React utilizes features of modern JavaScript for many of its patterns. Its biggest departure from JavaScript comes with the use of [JSX](https://reactjs.org/docs/introducing-jsx.html) syntax. JSX extends JavaScript's syntax so that HTML-like code can live alongside it. For example:

const heading = <h1>Mozilla Developer Network</h1>;

Copy to Clipboard

This heading constant is known as a **JSX expression**. React can use it to render that [<h1>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Heading_Elements) tag in our app.

Suppose we wanted to wrap our heading in a [<header>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/header) tag, for semantic reasons? The JSX approach allows us to nest our elements within each other, just like we do with HTML:

const header = (

<header>

<h1>Mozilla Developer Network</h1>

</header>

);

Copy to Clipboard

**Note:** The parentheses in the previous snippet aren't unique to JSX, and don't have any effect on your application. They're a signal to you (and your computer) that the multiple lines of code inside are part of the same expression. You could just as well write the header expression like this:

const header = <header>

<h1>Mozilla Developer Network</h1>

</header>

Copy to Clipboard

However, this looks kind of awkward, because the [<header>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/header) tag that starts the expression is not indented to the same position as its corresponding closing tag.

Of course, your browser can't read JSX without help. When compiled (using a tool like [Babel](https://babeljs.io/) or [Parcel](https://parceljs.org/)), our header expression would look like this:

const header = React.createElement("header", null,

React.createElement("h1", null, "Mozilla Developer Network")

);

Copy to Clipboard

It's possible to skip the compilation step and use [React.createElement()](https://reactjs.org/docs/react-api.html" \l "createelement" \t "_blank) to write your UI yourself. In doing this, however, you lose the declarative benefit of JSX, and your code becomes harder to read. Compilation is an extra step in the development process, but many developers in the React community think that the readability of JSX is worthwhile. Plus, modern front-end development almost always involves a build process anyway — you have to downlevel modern syntax to be compatible with older browsers, and you may want to [minify](https://developer.mozilla.org/en-US/docs/Glossary/Minification) your code to optimize loading performance. Popular tooling like Babel already comes with JSX support out-of-the-box, so you don't have to configure compilation yourself unless you want to.

Because JSX is a blend of HTML and JavaScript, some developers find it intuitive. Others say that its blended nature makes it confusing. Once you're comfortable with it, however, it will allow you to build user interfaces more quickly and intuitively, and allow others to better understand your codebase at a glance.

To read more about JSX, check out the React team's [JSX In Depth](https://reactjs.org/docs/jsx-in-depth.html) article.

## [Setting up your first React app](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#setting_up_your_first_react_app)

There are many ways to use React, but we're going to use the command-line interface (CLI) tool create-react-app, as mentioned earlier, which expedites the process of developing a React application by installing some packages and creating some files for you, handling the tooling described above.

It's possible to [add React to a website without create-react-app](https://reactjs.org/docs/add-react-to-a-website.html) by copying some [<script>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script) elements into an HTML file, but the create-react-app CLI is a common starting point for React applications. Using it will allow you to spend more time building your app, and less time fussing with setup.

### [Requirements](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#requirements)

In order to use create-react-app, you need to have [Node.js](https://nodejs.org/en/) installed. It's recommended that you use the long-term support (LTS) version. Node includes npm (the node package manager), and npx (the node package runner).

You may also use the Yarn package manager as an alternative, but we'll assume you are using npm in this set of tutorials. See [Package management basics](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Package_management) for more information on npm and yarn.

If you're using Windows, you will need to install some software to give you parity with Unix/macOS terminal in order to use the terminal commands mentioned in this tutorial. **Gitbash** (which comes as part of the [git for Windows toolset](https://gitforwindows.org/)) or [**Windows Subsystem for Linux**](https://docs.microsoft.com/windows/wsl/about) (**WSL**) are both suitable. See [Command line crash course](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Command_line) for more information on these, and on terminal commands in general.

Also bear in mind that React and ReactDOM produce apps that only work on a fairly modern set of browsers — IE9+ by way of some polyfills. It is recommended that you use a modern browser like Firefox, Microsoft Edge, Safari, or Chrome when working through these tutorials.

Also, see the following for more information:

* ["About npm" on the npm blog](https://docs.npmjs.com/about-npm/)
* ["Introducing npx" on the npm blog](https://blog.npmjs.org/post/162869356040/introducing-npx-an-npm-package-runner)
* [The create-react-app documentation](https://create-react-app.dev/)

### [Initializing your app](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#initializing_your_app)

create-react-app takes one argument: the name you'd like to give your app. create-react-app uses this name to make a new directory, then creates the necessary files inside it. Make sure you cd to the place you'd like your app to live on your hard drive, then run the following in your terminal:

npx create-react-app moz-todo-react

Copy to Clipboard

This creates a moz-todo-react directory, and does several things inside it:

* Installs some npm packages essential to the functionality of the app.
* Writes scripts for starting and serving the application.
* Creates a structure of files and directories that define the basic app architecture.
* Initializes the directory as a git repository, if you have git installed on your computer.

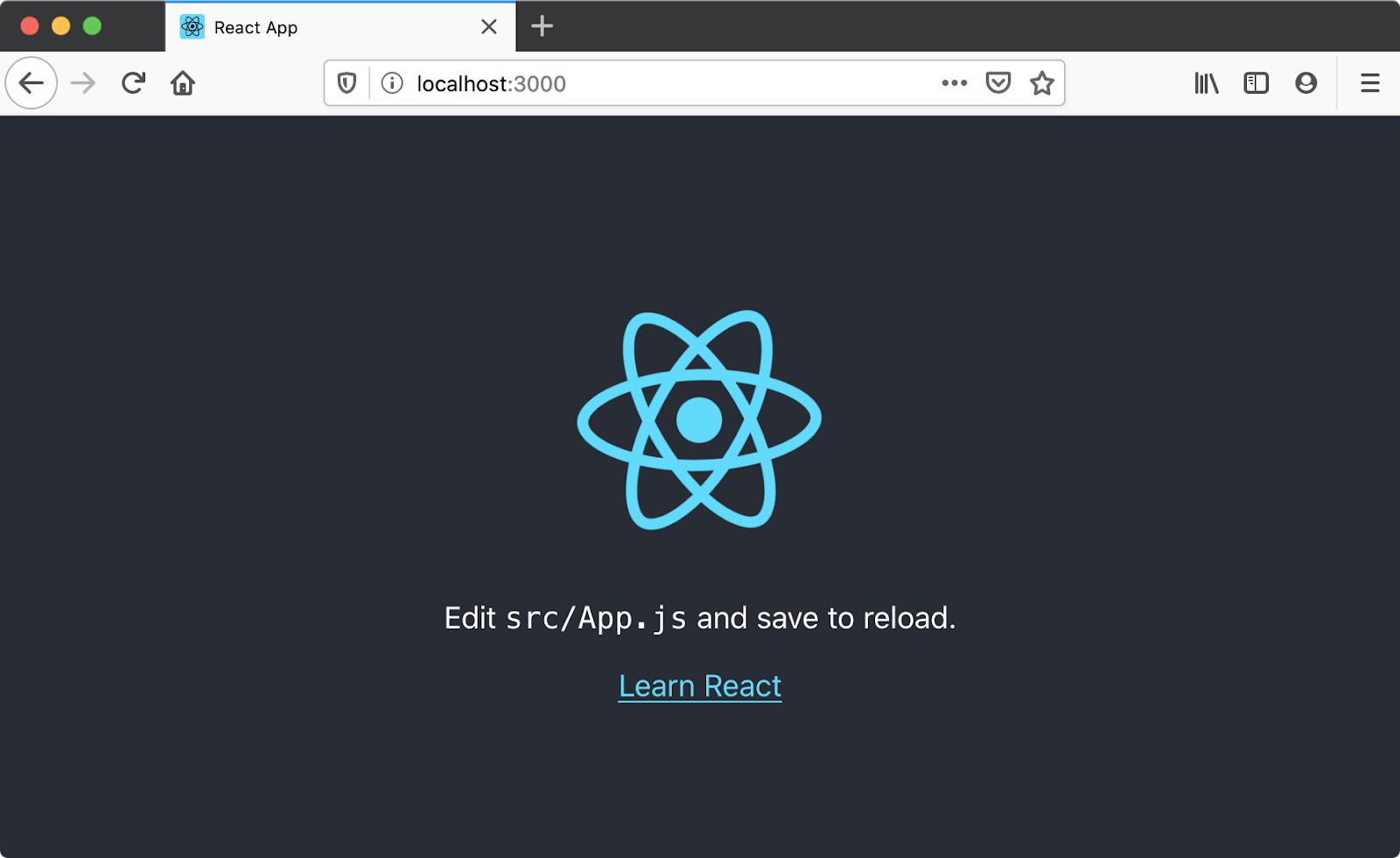
**Note:** if you have the Yarn package manager installed, create-react-app will default to using it instead of npm. If you have both package managers installed and explicitly want to use npm, you can add the flag --use-npm when you run create-react-app:

npx create-react-app moz-todo-react --use-npm

Copy to Clipboard

create-react-app will display a number of messages in your terminal while it works; this is normal! This might take a few minutes, so now might be a good time to go make a cup of tea.

When the process is complete, cd into the moz-todo-react directory and run the command npm start. The scripts installed by create-react-app will start being served at a local server at localhost:3000, and open the app in a new browser tab. Your browser will display something like this:



### [Application structure](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#application_structure)

create-react-app gives us everything we need to develop a React application. Its initial file structure looks like this:

moz-todo-react

├── README.md

├── node\_modules

├── package.json

├── package-lock.json

├── .gitignore

├── public

│ ├── favicon.ico

│ ├── index.html

│ ├── logo192.png

│ ├── logo512.png

│ ├── manifest.json

│ └── robots.txt

└── src

├── App.css

├── App.js

├── App.test.js

├── index.css

├── index.js

├── logo.svg

├── reportWebVitals.js

└── setupTests.js

The **src** directory is where we'll spend most of our time, as it's where the source code for our application lives.

The **public** directory contains files that will be read by your browser while you're developing the app; the most important of these is index.html. React injects your code into this file so that your browser can run it. There's some other markup that helps create-react-app function, so take care not to edit it unless you know what you're doing. You very much should change the text inside the [<title>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/title) element in this file to reflect the title of your application. Accurate page titles are important for accessibility!

The public directory will also be published when you build and deploy a production version of your app. We won't cover deployment in this tutorial, but you should be able to use a similar solution to that described in our [Deploying our app](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Deployment) tutorial.

The package.json file contains information about our project that Node.js/npm uses to keep it organized. This file is not unique to React applications; create-react-app merely populates it. You don't need to understand this file at all to complete this tutorial, however, if you'd like to learn more about it, you can read [package.json on the npm blog](https://docs.npmjs.com/cli/v9/configuring-npm/package-json/" \t "_blank); we also talk about it in our [Package management basics](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Package_management) tutorial.

## [Exploring our first React component — <App/>](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#exploring_our_first_react_component_%E2%80%94_app)

In React, a **component** is a reusable module that renders a part of our app. These parts can be big or small, but they are usually clearly defined: they serve a single, obvious purpose.

Let's open src/App.js, since our browser is prompting us to edit it. This file contains our first component, App, and a few other lines of code:

import logo from './logo.svg';

import './App.css';

function App() {

return (

<div className="App">

<header className="App-header">

<img src={logo} className="App-logo" alt="logo" />

<p>

Edit <code>src/App.js</code> and save to reload.

</p>

<a

className="App-link"

href="https://reactjs.org"

target="\_blank"

rel="noopener noreferrer"

>

Learn React

</a>

</header>

</div>

);

}

export default App;

Copy to Clipboard

The App.js file consists of three main parts: some [import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) statements at the top, the App component in the middle, and an [export](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export) statement at the bottom. Most React components follow this pattern.

### [Import statements](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#import_statements)

The import statements at the top of the file allow App.js to use code that has been defined elsewhere. Let's look at these statements more closely.

import logo from './logo.svg';

import './App.css';

Copy to Clipboard

The first statement imports a logo from './logo.svg'. Note the use of ./ at the beginning of the path and the .svg extension at the end — these tell us that the file is local and that it is not a JavaScript file. Indeed, the logo.svg file lives in our source directory.

The second statement imports the CSS related to our App component. Note that there is no variable name and no from directive. This is called a [side-effect import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import#import_a_module_for_its_side_effects_only) — it doesn't import any value into the JavaScript file, but it tells Webpack, the bundler, to add the referenced CSS file to the final CSS bundle.

Releases of React prior to the React 17 release in 2020 also required an import of the React library itself, as in - import React from 'react'. Skipping this step would result in an error: React turned the JSX we write into React.createElement(), so all React components needed to import the React module. React 17 introduced a new, rewritten version of the JSX transform that makes this statement unnecessary, with backported support to React 16.14.0, React 15.7.0, and React 0.14.10 (read more on the [official React doc](https://reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html)).

### [The App component](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#the_app_component)

After the imports, we have a function named App. Whereas most of the JavaScript community prefers camel-case names like helloWorld, React components use pascal-case variable names, like HelloWorld, to make it clear that a given JSX element is a React component, and not a regular HTML tag. If you were to rename the App function to app, your browser would show you an error.

Let's look at App more closely.

function App() {

return (

<div className="App">

<header className="App-header">

<img src={logo} className="App-logo" alt="logo" />

<p>

Edit <code>src/App.js</code> and save to reload.

</p>

<a

className="App-link"

href="https://reactjs.org"

target="\_blank"

rel="noopener noreferrer"

>

Learn React

</a>

</header>

</div>

);

}

Copy to Clipboard

The App function returns a JSX expression. This expression defines what your browser ultimately renders to the DOM.

Some elements in the expression have attributes, which are written just like in HTML, following a pattern of attribute="value". On line 3, the opening [<div>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div) tag has a className attribute. This is the same as the [class](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/class) attribute in HTML, but because JSX is JavaScript, we can't use the word class — it's reserved, meaning JavaScript already uses it for a specific purpose and it would cause problems here in our code. A few other HTML attributes are written differently in JSX than they are in HTML too, for the same kind of reason. We'll cover them as we encounter them.

Take a moment to change the [<p>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/p) tag on line 6 so that it reads "Hello, World!", then save your file. You'll notice that this change is immediately rendered in the development server running at http://localhost:3000 in your browser. Now delete the [<a>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/a) tag and save; the "Learn React" link will be gone.

Your App component should now look like this:

function App() {

return (

<div className="App">

<header className="App-header">

<img src={logo} className="App-logo" alt="logo" />

<p>

Hello, World!

</p>

</header>

</div>

);

}

Copy to Clipboard

### [Export statements](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#export_statements)

At the very bottom of the App.js file, the statement export default App makes our App component available to other modules.

## [Interrogating the index](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#interrogating_the_index)

Let's open src/index.js, because that's where the App component is being used. This file is the entry point for our app, and it initially looks like this:

import React from 'react';

import ReactDOM from 'react-dom/client';

import './index.css';

import App from './App';

import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

<React.StrictMode>

<App />

</React.StrictMode>

);

// If you want to start measuring performance in your app, pass a function

// to log results (for example: reportWebVitals(console.log))

// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals

reportWebVitals();

Copy to Clipboard

As with App.js, the file starts by importing all the JS modules and other assets it needs to run.

The first two statements import the React and ReactDOM libraries because they are referenced later in the file. We don't write a path or extension when importing these libraries because they are not local files. In fact, they are listed as dependencies in our package.json file. Be careful of this distinction as you work through this lesson!

index.css holds global styles that are applied to our whole app. We can also see our App component imported here; it is made available for import thanks to the export statement at the bottom of App.js.

Line 7 calls React's ReactDOM.createRoot() function with the DOM element inside which we want the component to be rendered, in this case the element with an ID of root. If you look inside public/index.html, you'll see that this is a <div> element just inside the <body>. React will create a root for this node, and take over managing the DOM inside it (read more on the [official react doc](https://beta.reactjs.org/apis/react-dom/client/createRoot)). The function returns the root which we can use to render a React element into the DOM.

Line 8 calls root.render() with the component we want to render, <App /> in this case.

All of this tells React that we want to render our React application with the App component as the root, or first component.

**Note:** In JSX, React components and HTML elements must have closing slashes. Writing just <App> or just <img> will cause an error.

[reportWebVitals](https://create-react-app.dev/docs/measuring-performance/) are a set of useful metrics that aim to capture the user experience of a web page, but they're not in scope for this article. You can delete its import line, as well as the reportWebVitals(); line.

Your final index.js file should look like this:

import React from 'react';

import ReactDOM from 'react-dom/client';

import './index.css';

import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'))

root.render(<App />);

Copy to Clipboard

## [Variables and props](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#variables_and_props)

Next, we'll use a few of our JavaScript skills to get a bit more comfortable editing components and working with data in React. We'll talk about how variables are used inside JSX, and introduce props, which are a way of passing data into a component (which can then be accessed using variables).

### [Variables in JSX](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#variables_in_jsx)

Back in App.js, let's focus on line 8:

<img src={logo} className="App-logo" alt="logo" />

Copy to Clipboard

Here, the <img /> tag's src attribute value is in curly braces. This is how JSX recognizes variables. React will see {logo}, know you are referring to the logo import on line 2 of our app, then retrieve the logo file and render it.

Let's try making a variable of our own. Before the return statement of App, add const subject = 'React';. Your App component should now look like this:

function App() {

const subject = "React";

return (

<div className="App">

<header className="App-header">

<img src={logo} className="App-logo" alt="logo" />

<p>

Hello, World!

</p>

</header>

</div>

);

}

Copy to Clipboard

Change line 8 to use our subject variable instead of the word "World", like this:

function App() {

const subject = "React";

return (

<div className="App">

<header className="App-header">

<img src={logo} className="App-logo" alt="logo" />

<p>

Hello, {subject}!

</p>

</header>

</div>

);

}

Copy to Clipboard

When you save, your browser should display "Hello, React!" instead of "Hello, World!"

Variables are convenient, but the one we've just set doesn't make great use of React's features. That's where props come in.

### [Component props](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#component_props)

A **prop** is any data passed into a React component. React props are comparable to HTML attributes. Where HTML elements have attributes, React components have props. Props are written inside component calls, and use the same syntax as HTML attributes — prop="value". In React, dataflow is unidirectional: props can only be passed from Parent components down to Child components; and props are read-only.

Let's open index.js and give our <App/> call its first prop.

Add a prop of subject to the <App/> component call, with a value of Clarice. When you are done, your code should look something like this:

root.render(<App subject="Clarice" />);

Copy to Clipboard

Back in App.js, let's revisit the App function itself, which reads like this (with the return statement shortened for brevity):

function App() {

const subject = "React";

return (

// return statement

);

}

Copy to Clipboard

Change the signature of the App function so that it accepts props as a parameter, and delete the subject const. Just like any other function parameter, you can put props in a console.log() to print it to your browser's console. Go ahead and do that before the return statement, like so:

function App(props) {

console.log(props);

return (

// return statement

);

}

Copy to Clipboard

With this change, {subject} becomes undefined, so comment out the line Hello, {subject}! for now. Save your file and check your browser's JavaScript console. You should see something like this logged:

Object { subject: "Clarice" }

The object property subject corresponds to the subject prop we added to our <App /> component call, and the string Clarice corresponds to its value. Component props in React are always collected into objects in this fashion.

Now that subject is one of our props, let's utilize it in App.js. Change the subject constant so that, instead of defining it as the string React, you are reading the value of props.subject. Now, you can also uncomment the line Hello, {subject}! and, if you wish, delete your console.log().

function App(props) {

const subject = props.subject;

return (

// return statement

);

}

Copy to Clipboard

When you save, the app should now greet you with "Hello, Clarice!". If you return to index.js, edit the value of subject, and save, your text will change. Note that if you wanted to leave in the Hello line throughout this change, you could also have updated the JSX variable to {props.subject}.

# React resources

* [Previous](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_accessibility)
* [Overview: Client-side JavaScript frameworks](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks)
* [Next](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Ember_getting_started)

Our final article provides you with a list of React resources that you can use to go further in your learning.

|  |  |
| --- | --- |
| **Prerequisites:** | Familiarity with the core [HTML](https://developer.mozilla.org/en-US/docs/Learn/HTML), [CSS](https://developer.mozilla.org/en-US/docs/Learn/CSS), and [JavaScript](https://developer.mozilla.org/en-US/docs/Learn/JavaScript) languages, knowledge of the [terminal/command line](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Command_line). |
| **Objective:** | To provide further resources for learning more about React. |

## [Component-level styles](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_resources#component-level_styles)

Although this tutorial doesn't use this approach, many React applications define their styles on a per-component basis, rather than in a single, monolithic stylesheet.

create-react-app makes it possible to import CSS files into JavaScript modules, so that CSS is only sent to your user when the corresponding component is rendered. For this app, we could have for example written a dedicated Form.css file to house the styles of those respective components, then imported the styles into their respective modules like this:

import Form from "./Form";

import "./Form.css";

Copy to Clipboard

This approach makes it easy to identify and manage the CSS that belongs to a specific component. However, it also fragments your stylesheet across your codebase, and this fragmentation might not be worthwhile. For larger applications with hundreds of unique views and lots of moving parts, it makes sense to limit the amount of irrelevant code that's sent to your user. You'll likely have app-wide styles and specific component styles that build on top of those.

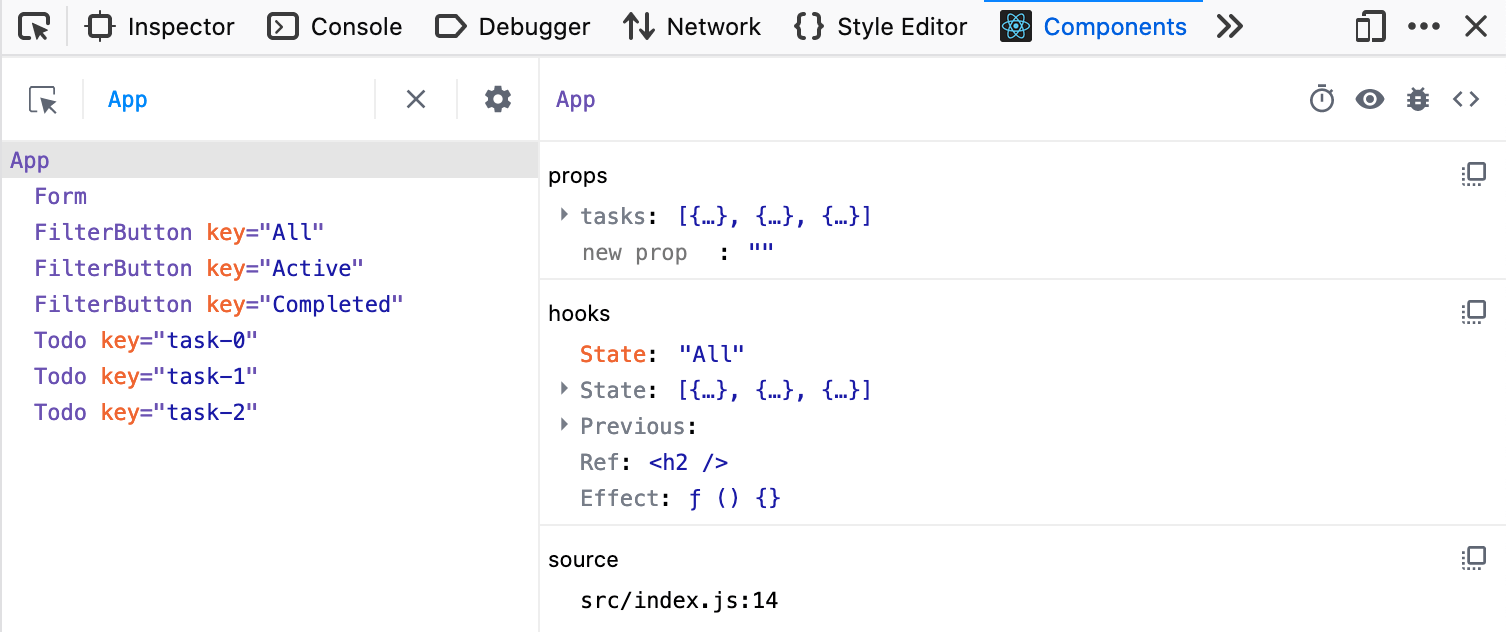
You can [read more about component stylesheets in the create-react-app docs](https://create-react-app.dev/docs/adding-a-stylesheet/).

## [React DevTools](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_resources#react_devtools)

We used console.log() to check on the state and props of our application in this tutorial, and you'll also have seen some of the useful warnings and error message that React gives you both in the CLI and your browser's JavaScript console. But there's more we can do here.

The React DevTools utility allows you to inspect the internals of your React application directly in the browser. It adds a new panel to your browser's developer tools, and with it you can inspect the state and props of various components, and even edit state and props to make immediate changes to your application.

This screenshot shows our finished application as it appears in React DevTools:



On the left, we see all of the components that make up our application, including some unique keys for the things that are rendered from arrays. On the right, we see the props and hooks that our App component utilizes. Notice, too, that the Form, FilterButton, and Todo components are indented to the right – this indicates that App is their parent. In more complex apps, this view is great for understanding parent/child relationships at a glance.

React DevTools is available in a number of forms:

* A [Chrome browser extension](https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en).
* A [Firefox browser extension](https://addons.mozilla.org/en-US/firefox/addon/react-devtools/).
* A [Microsoft Edge browser extension](https://microsoftedge.microsoft.com/addons/detail/react-developer-tools/gpphkfbcpidddadnkolkpfckpihlkkil).
* A [stand-alone application you can install with npm or Yarn](https://www.npmjs.com/package/react-devtools).

Try installing one of these, then using it to inspect the app you've just built!

You can [read more about React DevTools on the React blog](https://reactjs.org/blog/2019/08/15/new-react-devtools.html).

## [The Context API](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_resources#the_context_api)

The application that we built in this tutorial utilized component props to pass data from its App component to the child components that needed it. Most of the time, props are an appropriate method for sharing data; for complex, deeply nested applications, however, they're not always best.

React provides the [Context API](https://reactjs.org/docs/context.html) as a way to provide data to components that need it without passing props down the component tree. There's also [a useContext hook](https://reactjs.org/docs/hooks-reference.html#usecontext) that facilitates this.

If you'd like to try this API for yourself, Smashing Magazine has written an [introductory article about React context](https://www.smashingmagazine.com/2020/01/introduction-react-context-api/).

## [Class components](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_resources#class_components)

Although this tutorial doesn't mention them, it is possible to build React components using classes – these are called class components. Until the arrival of hooks, classes were the only way to bring state into components or manage rendering side effects. They're still the only way to handle certain other, more edge-case features, and they're very common in legacy React projects. The official React docs are a great place to start learning about them.

* [State and Lifecycle in the React Docs](https://reactjs.org/docs/state-and-lifecycle.html)
* [Intro To React in the React Docs](https://reactjs.org/tutorial/tutorial.html)
* [Read about JavaScript classes at MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes)

## [Testing](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_resources#testing)

create-react-app provides some tools for testing your application out of the box — you may have deleted the relevant files earlier in the tutorial. The documentation for create-react-app [covers some basics for testing](https://create-react-app.dev/docs/running-tests/).

## [Routing](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_resources#routing)

While routing is traditionally handled by a server and not an application on the user's computer, it is possible to configure a web application to read and update the browser's location, and render certain user interfaces. This is called client-side routing. It's possible to create many unique routes for your application (such as /home, /dashboard, or login/).

The React community has produced two major libraries for client-side routing: [React Router](https://reactrouter.com/) and [Reach Router](https://reach.tech/router/).

* React Router is well-suited to applications with complex routing needs, and it meets some edge cases better than Reach Router. React Router is a larger library, however.
* Reach Router is well-suited to simpler applications, and automatically manages focus as the user navigates from page to page.

Focus management is essential in client-side routing — without it, keyboard users can be trapped in focus limbo, and screen-reader users may have no idea that they have moved to a new page. Because Reach Router is better for accessibility, it's a good place to start.

There's one caveat, however: these projects will be [merging in the near future](https://reacttraining.com/blog/reach-react-router-future/). When this merge happens, React Router will be the surviving project (with the addition of the focus management features of Reach).

## [Summary](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started#summary)

This brings us to the end of our initial look at React, including how to install it locally, creating a starter app, and how the basics work. In the next article, we'll start building our first proper application — a todo list. Before we do that, however, let's recap some of the things we've learned.

In React:

* Components can import modules they need and must export themselves at the bottom of their files.
* Component functions are named with PascalCase.
* You can read JSX variables by putting them between curly braces, like {so}.
* Some JSX attributes are different than HTML attributes so that they don't conflict with JavaScript reserved words. For example, class in HTML translates to className in JSX. Note that multi-word attributes are in camelCase.
* Props are written just like attributes inside component calls and are passed into components.
* [Previous](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Main_features)
* [Overview: Client-side JavaScript frameworks](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks)

### 