HTML Course

Flavio Copes

Updated August 3, 2026

Contents

Learn HTML by building a small multi-page website with semantic structure, links, images, tables, accessibility checks, and inline quizzes.

What you'll learn: Build a small accessible website, then inspect, validate, and fix its HTML.

How a web page reaches you

The Web, browsers, servers, URLs, and the role HTML plays.

What is the Web?

Learn what the Web is, how it differs from the Internet, and where HTML fits when you open a page in your browser.

The Web is a collection of documents and applications connected with links and reached through URLs.

The Internet is the network underneath it. It connects computers around the world. The Web is one of the things we use that network for, just like email is another.

When you visit a website, two programs take part in the exchange:

- a **browser**, such as Chrome, Firefox, or Safari
- a **server**, a program that waits for requests and sends back responses

The browser asks for a resource. The server responds with that resource. Very often the response contains HTML.

```
browser  -- request -->  server
browser  <-- response -- server
```

HTML describes the content and structure of a page. CSS controls its presentation. JavaScript can add behavior.

Those three technologies work together, but HTML comes first. A page can work without CSS or JavaScript. Without HTML, there is no document for the browser to show.

In this course we'll focus on the document. You will learn what each part means and how to choose elements that describe your content clearly.

Try it

Open any page, right-click, and choose **View Page Source**. What you see is the HTML response the browser received. It may look busy. That's fine. By the end of this course, its structure will make much more sense.

Browsers and servers

Follow the simple request and response exchange between a browser and a web server before the browser renders an HTML page.

A browser is an HTTP client. You give it a URL, and it makes a request on your behalf.

The server receives that request and decides what to send back. It might read an HTML file from disk. It might generate HTML using data from a database. The browser does not need to know which approach the server used.

> **Diagram:** A browser sends a request to a server, and the server returns an HTML response.

A simplified exchange looks like this:

```
GET /about/ HTTP/1.1
Host: example.com
```

The server could answer:

```
HTTP/1.1 200 OK
Content-Type: text/html

<h1>About us</h1>
```

The browser reads the `Content-Type` header and knows the response contains HTML. It parses the markup and turns it into a document on the screen.

This distinction is useful:

- the server **sends** HTML
- the browser **interprets** HTML

The browser also provides default styles. This is why a heading appears larger than a paragraph even before you write any CSS.

HTML is forgiving. Browsers try hard to display imperfect documents. That helps the Web keep working, but it also means a page that looks correct can still contain mistakes. Later we'll use validation tools to find them.

Read a URL

Break a web address into its scheme, host, path, query, and fragment so links and resources make sense later in the course.

A URL identifies a resource on the Web.

Consider this one:

`https://example.com/guides/html/?format=short#links`

It has several parts:

- `https` is the **scheme**
- `example.com` is the **host**
- `/guides/html/` is the **path**
- `format=short` is the **query string**
- `links` is the **fragment**

The scheme tells the browser which protocol to use. For websites, that is normally HTTPS.

The host identifies the server. The path identifies a resource on that server. A query string passes extra information, while the fragment points to a location inside the document.

There is one important detail: the browser does not send the fragment in the HTTP request. It uses the fragment after the page arrives, usually to scroll to an element with a matching `id`.

You will use URLs in links, images, stylesheets, scripts, audio, video, and forms. Understanding their parts now will make all of those elements easier to use.

Try it

Take the URL in your address bar and identify its scheme, host, and path. If it has a `?` or `#`, identify the query and fragment too.

What HTML does

Understand HTML as a markup language that gives content structure and meaning rather than deciding how the page should look.

HTML stands for HyperText Markup Language.

It is a **markup language**. You start with content and mark each part according to what it means.

```
<h1>My travel notes</h1>
<p>I spent three days in Copenhagen.</p>
```

The tags tell the browser that the first line is the main heading and the second is a paragraph.

Most elements have an opening tag, some content, and a closing tag:

```
<p>A paragraph of text</p>
```

That whole thing is a `p` **element**. `<p>` and `</p>` are its tags.

Some elements cannot contain content and do not have a closing tag. The image element is one example:

```
<img src="photo.jpg" alt="A bicycle beside a canal">
```

HTML is not concerned with how the content looks. You do not use it to say “make this heading blue.” That is CSS's job.

Instead, HTML answers questions such as:

- Is this text a heading or a paragraph?
- Is this a link?
- Is this a list of related items?
- Is this the main navigation?

Choosing elements for their meaning makes a page easier to use with browsers, search engines, screen readers, and other tools.

Practice: trace a page request

Follow one real page request in your browser and identify the URL, server response, content type, and HTML source.

Let's turn the request and response into something you can see.

Open a page in your browser. You can use this course page.

Then open the browser developer tools and select the **Network** panel. Reload the page. You will see a list of requests.

Find the first request whose type is `document`. Click it and look for these details:

- the request URL
- the request method
- the response status
- the `Content-Type` response header

You should find a GET request, a successful status such as `200`, and a content type that includes `text/html`.

Now open the response tab. This is the HTML the server sent to the browser.

The browser made more requests after receiving it. Those requests fetch stylesheets, scripts, images, and fonts mentioned by the document.

Draw the exchange

Write this on paper or in a text file:

```
My browser asked for:

The server answered with:

The content type was:

The HTML caused these extra requests:
```

Fill each line with something you observed.

You do not need to understand every header. The goal is to connect the words **browser**, **request**, **server**, **response**, and **HTML** to a real exchange.

Done when

You are done when you can point to the document request and explain which side sent the HTML.

Your first document

Create a complete HTML document with useful metadata.

Create an HTML file

Create a small HTML file, open it directly in a browser, and make your first edit without needing a framework or web server.

You only need a text editor and a browser to create an HTML page.

Create a new folder named `html-course`. Inside it, create a file named `index.html`.

Add this markup:

```
<h1>My first page</h1>
<p>Hello from an HTML file.</p>
```

Save the file, then open it in your browser. You can double-click it in your file manager or drag it into a browser window.

The address will start with `file://` instead of `https://`. That is expected. The browser is reading the document from your computer rather than requesting it from a web server.

Change the paragraph, save the file, and refresh the page. This edit-save-refresh loop is the simplest web development setup you can have.

The `.html` extension matters. It tells your operating system and tools that this is an HTML document.

Our tiny example works, but it is only a fragment. A complete HTML document contains a few more elements. We'll add them in the next lesson.

The document structure

Build a complete HTML document with the doctype, root html element, head metadata, and visible body content in the right places.

A complete HTML page starts with this structure:

Diagram: HTML is a tree. Every element has a place in it.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My first page</title>
  </head>
  <body>
    <h1>My first page</h1>
    <p>Hello from an HTML document.</p>
  </body>
```

```
</html>
```

The `html` element is the root element. Every other element belongs inside it.

Inside `html` there are two main parts:

- `head` contains information **about** the document
- `body` contains the document people see and use

A document has one `html`, one `head`, and one `body` element.

Notice the indentation. Elements inside `html` are indented by two spaces. Elements inside `head` and `body` are indented again.

Indentation does not normally change what the browser displays. It shows the nesting clearly to humans, which makes errors much easier to spot.

Replace the fragment in your `index.html` file with this complete document. Save and refresh. The visible result stays almost the same, but the browser now has a properly described page.

Take this interactive quiz online

The doctype

See why every HTML document begins with a short doctype declaration and how it keeps browsers in standards mode.

The first line in a modern HTML document is:

```
<!doctype html>
```

This is the document type declaration, usually called the **doctype**.

It tells the browser to render the page using standards mode. Without a recognized doctype, browsers can enter quirks mode, which imitates some old browser behavior so ancient pages do not break.

You do not want quirks mode for a new page.

The doctype is not an HTML element. It has no closing tag and does not belong inside the `html` element. Put it on the first line, before everything else.

You might see very long doctypes in old documents. Modern HTML only needs this short version. HTML is maintained as a living standard, so you do not need to put a version number in it.

Uppercase and lowercase both work, but this is the form I use:

```
<!doctype html>
```

The head of the document

Add a useful title, character encoding, and viewport metadata to the part of the document browsers read before showing the page.

The `head` contains information about the document. Most of it does not appear inside the page.

Start with these three elements:

```
<head>
  <meta charset="utf-8">
```

```
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>My first page</title>
</head>
```

`charset="utf-8"` tells the browser how the document's characters are encoded. UTF-8 supports the characters you will normally need, including emoji. Put this declaration near the start of `head`.

The viewport meta element tells mobile browsers to use the device width as the page width. Without it, a responsive page can look like a wide desktop page squeezed onto a small screen.

The `title` gives the document a name. Browsers show it in the tab. Search engines often use it as the linked title in search results. Bookmark tools use it too.

The title should describe the specific page. “About Acme” is more useful than “Page.”

You can also load CSS and JavaScript from the head. We will cover those in their own courses. For now, build a solid minimum head and keep moving.

Take this interactive quiz online

Language and description

Declare the page language and write a concise meta description that helps assistive technology and search results understand the document.

Add the document language to the opening `html` tag:

```
<html lang="en">
```

The `lang` attribute helps screen readers choose the correct pronunciation rules. It also helps translation and search tools understand the content.

Use the language the page is actually written in. For Italian, use `it`. For Brazilian Portuguese, use `pt-BR` when that regional distinction matters.

The head can also contain a short description:

```
<meta
  name="description"
  content="Notes from a three-day cycling trip around Copenhagen."
>
```

The description does not appear in the page body. Search engines may show it below the page title, although they can choose a different excerpt.

Write it for a person deciding whether the page answers their question. Keep it specific. Avoid lists of keywords.

Your document now has a useful foundation:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta name="description" content="Notes from a cycling trip.">
    <title>Copenhagen cycling notes</title>
  </head>
```

```
  <body>
    <h1>Copenhagen cycling notes</h1>
  </body>
</html>
```

Practice: build your page shell

Create the complete document shell for a personal page with a doctype, language, useful title, description, and visible content.

We will build one small personal page through the rest of the course.

Create a folder named `my-page`. Inside it, create `index.html`.

Start with this document:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Sam's web page</title>
    <meta
      name="description"
      content="Projects, notes, and useful links from Sam."
    >
  </head>
  <body>
    <h1>Sam's web page</h1>
    <p>I am learning how the Web works.</p>
  </body>
</html>
```

Replace Sam's name and description with your own.

Open the file in your browser. The heading and paragraph appear inside the page. The title appears in the browser tab.

Check the invisible parts

Use **View Page Source**. Confirm that the browser received the same structure you wrote.

Then check these details:

- `doctype` is the first line
- `html` has the correct language
- `head` contains the title and metadata
- `body` contains the visible page
- every opened element is closed

Save this file. Each practice lesson will improve it.

Done when

You have a valid page shell that opens locally and has a useful browser-tab title.

Text and meaning

Structure readable text with the appropriate elements.

Headings and paragraphs

Organize a document with a clear heading hierarchy and paragraphs that represent real blocks of related text.

Headings give a document its outline. HTML provides six levels, from `h1` to `h6`.

```
<h1>Copenhagen cycling guide</h1>

<h2>Choose a bicycle</h2>
<p>A city bike is enough for most routes.</p>

<h2>Plan your route</h2>
<h3>Routes for beginners</h3>
<p>Start with the separated paths around the lakes.</p>
```

The number describes the heading's level, not its desired font size. Do not choose `h4` because it looks smaller. CSS controls appearance.

Use `h1` for the page's main heading. Major sections use `h2`. A subsection inside an `h2` section uses `h3`, and so on.

Avoid jumping from `h2` to `h4`. A person scanning visually might not notice, but someone navigating with a screen reader depends on that structure.

The `p` element represents a paragraph:

```
<p>The route follows the harbor for five kilometers.</p>
```

Do not use repeated line breaks to create paragraphs. Use one `p` for each actual paragraph and let CSS control the spacing later.

Take this interactive quiz online

Emphasis and importance

Mark words as emphasized or important with semantic text elements instead of choosing tags only for their default visual style.

The `em` element adds emphasis. It can change the meaning of a sentence.

```
<p>You should <em>always</em> wear a helmet.</p>
```

Browsers display `em` as italic by default, but the meaning matters more than the style.

The `strong` element marks serious importance, urgency, or warning:

```
<p><strong>Do not leave your bicycle unlocked.</strong></p>
```

Browsers display `strong` as bold by default.

HTML also has `i` and `b`. They do not mean exactly the same thing:

- `em` is stress emphasis
- `strong` is importance

- `i` marks text set apart from normal prose, such as a foreign phrase or technical term
- `b` draws attention without adding importance, such as a keyword in a summary

Ask what the text means before choosing the element. If you only want italic or bold styling, CSS is usually the right tool.

Take this interactive quiz online

Lists

Use unordered, ordered, and description lists to group related items and communicate whether their sequence matters.

Use an unordered list when the order of its items does not matter:

```
<ul>
  <li>Helmet</li>
  <li>Water bottle</li>
  <li>Repair kit</li>
</ul>
```

`ul` means unordered list. Each item is an `li` element.

Use an ordered list when the sequence matters:

```
<ol>
  <li>Check the tires</li>
  <li>Adjust the seat</li>
  <li>Test the brakes</li>
</ol>
```

The browser numbers ordered-list items by default. The numbers are useful, but the semantic fact that order matters is even more important.

Description lists pair terms with descriptions:

```
<dl>
  <dt>Cycle track</dt>
  <dd>A path reserved for bicycles.</dd>

  <dt>Bike lane</dt>
  <dd>A marked part of a road for bicycles.</dd>
</dl>
```

You can nest a list inside a list item. Keep the nesting visible with consistent indentation.

Take this interactive quiz online

Quotes, code, and line breaks

Represent quotations, code, preserved whitespace, and intentional line breaks with elements made for those kinds of content.

Use `blockquote` for a quotation that forms its own block:

```
<blockquote>
  <p>The bicycle is a simple solution to some of the world's most complicated problems.</p>
```

```
</blockquote>
```

Use `q` for a short quotation inside a sentence:

```
<p>The guide called this route <q>the quiet way into town</q>.</p>
```

Use `code` for computer code:

```
<p>The page starts with the <code>&lt;!doctype html&gt;</code> declaration.</p>
```

For a block of code, combine `pre` and `code`:

```
<pre><code>&lt;h1&gt;Hello&lt;/h1&gt;
&lt;p&gt;Welcome to my page.&lt;/p&gt;</code></pre>
```

`pre` preserves spaces and line breaks. Normal HTML collapses sequences of whitespace into a single space.

The `br` element creates a line break within content where the break is meaningful, such as an address or a poem:

```
<p>
  42 Harbor Street<br>
  Copenhagen
</p>
```

Do not use a row of `br` elements to create vertical spacing. That is a presentation decision for CSS.

You can leave comments for yourself or other developers:

```
<!-- Replace this copy before launch -->
```

Comments do not appear in the rendered page, but they are visible in the source.

Take this interactive quiz online

Practice: add readable content

Turn the empty personal page into a readable document using headings, paragraphs, lists, emphasis, a quotation, and code.

Open the `index.html` file you created in the previous practice.

Keep the existing `h1`. Under it, add two sections of content using `h2` headings.

The first section can introduce you:

```
<h2>About me</h2>
<p>
  I am learning <strong>HTML</strong> so I can build pages
  that work for everyone.
</p>
```

The second can list what you are learning:

```
<h2>What I am learning</h2>
<ul>
  <li>How browsers request a page</li>
  <li>How HTML describes content</li>
  <li>How to choose meaningful elements</li>
```

```
</ul>
```

Add one short quotation you like. Use `blockquote` for the quotation and `cite` for its source.

Then mention one HTML element inside a paragraph. Use `code` around the element name:

```
<p>The <code>h1</code> element contains the page heading.</p>
```

Use your own words. The example is a starting point, not a template you must copy.

Read the outline

Ignore the visual size of the text for a moment. Read only the headings:

1. page title
2. About me
3. What I am learning

Do they describe the page in the right order? If not, fix the headings before changing anything else.

Done when

Your page is understandable as plain text and uses elements for meaning, not appearance.

Links and attributes

Connect documents and describe elements with attributes.

Attributes

Add information to an element with attributes and understand quoted values, boolean attributes, classes, and unique IDs.

Attributes add information to an element. They are written in the opening tag.

```
<p class="introduction">Welcome to the guide.</p>
```

Here, `class` is the attribute name and `introduction` is its value.

An element can have several attributes:

```
<p id="welcome" class="introduction featured">
  Welcome to the guide.
</p>
```

Quote attribute values. Double quotes are the common convention.

The `class` attribute can contain several space-separated names. You can reuse those names on many elements. CSS and JavaScript often use classes to find related elements.

An `id` identifies one element in the document. Keep every ID unique.

Some attributes are boolean. Their presence means true:

```
<button disabled>Save</button>
```

You will meet element-specific attributes throughout this course: `href` on links, `src` and `alt` on images, and many more.

Attributes describe or configure an element. They do not replace its content. A link still needs useful link text, and an image still needs a useful text alternative when it conveys information.

Create links

Create clear hyperlinks with the anchor element, an href destination, and link text that makes sense outside its surrounding sentence.

Links are what make the Web a web.

Create one with the anchor element, `a`, and the `href` attribute:

```
<a href="https://example.com/guides/cycling/">Read the cycling guide</a>
```

The `href` value is the destination. The content between the tags is the link text.

Write text that describes where the link goes. “Read the cycling guide” is more useful than “click here,” especially when someone scans a list of links with assistive technology.

An anchor without `href` is not a hyperlink:

```
<a>Not a working link</a>
```

A link can contain an image or other phrasing content, but it cannot contain another link.

```
<a href="/">
  <img src="logo.svg" alt="Acme home">
</a>
```

When an image is the only content of a link, its alt text should describe the link's purpose.

Absolute and relative URLs

Link between pages using full URLs, root-relative paths, and document-relative paths without accidentally pointing to the wrong location.

An absolute URL includes the scheme and host:

```
<a href="https://example.com/about/">About Example</a>
```

Use an absolute URL when you link to another website.

For pages on the same site, you can use a relative URL. A path that starts with `/` begins at the root of the current site:

```
<a href="https://flaviocopes.com/about/">About us</a>
```

This points to `/about/` no matter which page contains the link.

A path without the first slash is relative to the current document:

```
<a href="photos/">Photos</a>
```

If the current page is `https://example.com/trips/copenhagen/`, that link goes to:

```
https://example.com/trips/copenhagen/photos/
```

Use `../` to move up one directory:

```
<a href="../">All trips</a>
```

Relative URLs are useful because they keep working when the whole site moves to another domain.

The tradeoff is that you need to understand which document they are relative to.

Take this interactive quiz online

Fragments, email, and phone links

Link to a specific part of a page and create email or telephone actions with fragment, mailto, and tel destinations.

A fragment link points to an element in a document.

Give the destination an ID:

```
<h2 id="pricing">Pricing</h2>
```

Then link to it with `#pricing`:

```
<a href="#pricing">Skip to pricing</a>
```

The browser scrolls to the element and updates the URL. You can also add a fragment to another page:

```
<a href="https://flaviocopes.com/guide/#pricing">Read about pricing</a>
```

IDs must be unique so the browser knows which element to find.

The `mailto:` scheme opens the visitor's configured email application:

```
<a href="mailto:hello@example.com">Email us</a>
```

The `tel:` scheme offers to call a number on devices that support calls:

```
<a href="tel:+4532123456">Call +45 32 12 34 56</a>
```

These links trigger an action rather than navigating to an HTTP page. Make that action clear in the link text.

Take this interactive quiz online

Target, rel, and downloads

Decide when a link should open a new tab, describe its relationship, or suggest downloading a resource instead of navigating to it.

Links normally open in the current tab. That is a good default because it leaves navigation under the visitor's control.

When a new tab is genuinely useful, use `target="_blank"`:

```
<a
  href="https://example.com/report/"
  target="_blank"
  rel="noopener"
>
  Open the external report in a new tab
</a>
```

`rel` describes the relationship between the current page and the destination. `noopener` prevents the new page from receiving access to the page that opened it in older browser behavior. Modern browsers generally provide this protection for `_blank`, but stating it is harmless and explicit.

Opening a new tab can surprise people. If you do it, say so in the link text or nearby context.

The `download` attribute suggests downloading a same-origin resource:

```
<a href="https://flaviocopes.com/files/route-map.pdf" download>Download the route map</a>
```

The browser and server still have the final say. Cross-origin links may ignore the attribute, and response headers can affect the result.

Take this interactive quiz online

Practice: connect your pages

Add a second page and connect it with relative links, a page fragment, an email link, and useful link text.

Your page needs somewhere to go.

Create `about.html` beside `index.html`. Give it the same complete document structure, then add a heading and one paragraph about why you are learning HTML.

Link the two pages:

```
<a href="about.html">About me</a>
```

On `about.html`, add the return link:

```
<a href="index.html">Back to the home page</a>
```

These are relative URLs. Both files live in the same folder, so the filename is enough.

Add a page fragment

Give the learning section on `index.html` an ID:

```
<h2 id="learning">What I am learning</h2>
```

Now link directly to it:

```
<a href="#learning">Jump to what I am learning</a>
```

Add an email link if you want visitors to contact you:

```
<a href="mailto:sam@example.net">Email me</a>
```

Use your real address only if you are comfortable publishing it.

Test every path

Click each link. Do not assume it works because the HTML looks correct.

Check the address bar when you move between pages. Notice how a relative URL becomes a complete URL after the browser resolves it.

Avoid link text such as “click here.” The text should describe the destination.

Done when

You can move between both pages, jump to the learning section, and explain why each `href` has its current value.

Page structure

Give a page meaningful structure with semantic elements.

Page landmarks

Structure the major regions of a page with header, nav, main, and footer elements that give browsers and assistive tools useful meaning.

Semantic container elements describe the major regions of a page.

Diagram: Landmarks give the page a useful map.

```
<body>
  <header>
    <a href="/">City Cycles</a>
    <nav aria-label="Primary">
      <a href="https://flaviocopes.com/routes/">Routes</a>
      <a href="https://flaviocopes.com/repairs/">Repairs</a>
    </nav>
  </header>

  <main>
    <h1>Copenhagen cycling guide</h1>
    <p>Plan a safe route across the city.</p>
  </main>

  <footer>
    <p>Published by City Cycles.</p>
  </footer>
</body>
```

`header` contains introductory content. A page can also have headers inside articles and sections.

`nav` contains a major group of navigation links. You do not need to wrap every small set of links in `nav`.

`main` contains the unique main content of the page. A page should have one visible `main` element.

`footer` contains closing information for its nearest section or for the page as a whole.

These elements mostly behave like generic containers visually. Their value is the meaning they provide. Screen-reader users can use landmarks to jump directly to navigation or main content.

Take this interactive quiz online

Article, section, and aside

Choose between article, section, and aside by asking whether content stands alone, belongs to a titled section, or is secondary.

Use `article` for a self-contained composition that could make sense on its own.

```
<article>
  <h2>A winter route around the lakes</h2>
  <p>This route stays on cleared cycle tracks.</p>
```

```
</article>
```

Blog posts, news stories, forum posts, and product cards can be articles.

Use `section` for a thematic part of a document. It should normally have a heading.

```
<section>
  <h2>Routes for beginners</h2>
  <p>Start with short, separated paths.</p>
</section>
```

Do not use `section` just because you need a wrapper for CSS. If the content has no natural heading or theme, a `div` may be more honest.

Use `aside` for content that is related but secondary to the surrounding content:

```
<aside>
  <h2>Renting a bicycle</h2>
  <p>Most train stations have a rental shop nearby.</p>
</aside>
```

An aside inside an article relates to that article. An aside beside the page's main content often acts as a sidebar.

The right question is not “Which element looks right?” These elements have no important default visual difference. Ask what role the content plays.

Take this interactive quiz online

Div, span, and nesting

Use generic containers when no semantic element fits, and keep the document tree valid by closing and nesting elements carefully.

`div` and `span` are generic containers. They do not describe a specific kind of content.

Use `div` to group a block of content when no semantic element fits:

```
<div class="route-summary">
  <p>Distance: 8 km</p>
  <p>Difficulty: easy</p>
</div>
```

Use `span` around a small piece of phrasing content:

```
<p>Difficulty: <span class="rating">easy</span></p>
```

There is nothing wrong with these elements. The problem is using them when an element with useful meaning already exists. Prefer `button` for an action, `nav` for major navigation, and `p` for a paragraph.

HTML elements form a tree. Close inner elements before outer ones:

```
<p>This is <strong>very important</strong>.</p>
```

This is incorrectly crossed:

```
<p>This is <strong>incorrect.</p></strong>
```

Some combinations are invalid even when the tags look balanced. You cannot nest a link inside

another link, and a paragraph cannot contain a `div`.

Browsers try to repair invalid nesting. The resulting document tree may differ from the source you wrote, which can cause confusing CSS, JavaScript, and accessibility problems.

Take this interactive quiz online

Practice: structure your page

Reorganize the personal page with semantic landmarks and verify that every piece of content belongs in the right container.

Your page has content and links. Now give it a useful structure.

Start by placing the page title and navigation inside `header`:

```
<header>
  <h1>Sam's web page</h1>
  <nav aria-label="Primary">
    <a href="index.html">Home</a>
    <a href="about.html">About me</a>
  </nav>
</header>
```

Wrap the unique page content in `main`:

```
<main>
  <section>
    <h2>About me</h2>
    <p>I am learning how the Web works.</p>
  </section>

  <section id="learning">
    <h2>What I am learning</h2>
    <!-- your list goes here -->
  </section>
</main>
```

Finish with a small footer:

```
<footer>
  <p>Built while learning HTML.</p>
</footer>
```

Do not add an element because its name sounds useful. Add it because the content matches its meaning.

You probably do not need `article` or `aside` yet. That is fine. A smaller honest structure is better than a page filled with unnecessary containers.

Inspect the landmarks

Open the browser accessibility inspector if it is available. Look for `banner`, `navigation`, `main`, and `contentinfo` landmarks.

The HTML elements create those landmarks without extra ARIA roles.

Done when

Your page has one visible `main`, clear navigation, and no container you cannot explain.

Images and media

Add accessible and responsive images, audio, and video.

Add images

Add an image with a useful source and text alternative, and understand why the img element does not have a closing tag.

Use the `img` element to add an image:

```
<img src="bicycle.jpg" alt="A red bicycle parked beside a canal">
```

`src` tells the browser where to find the image. It accepts the same kinds of URLs you used in links.

`img` is a void element. It cannot contain content, so it has no closing tag.

The browser makes a separate request for the image after it parses the HTML. If the file is missing or the path is wrong, the image cannot appear.

Common image formats include JPEG, PNG, WebP, AVIF, GIF, and SVG. The right format depends on the image and how you produce it. That choice affects file size and quality, but it does not change the basic HTML.

The `alt` attribute is not optional decoration. It provides a text alternative when the image cannot be seen. We will focus on writing it well in the next lesson.

Try it

Put an image in the same folder as your HTML file. Add it with a relative URL, save, and refresh. Then deliberately change one letter in `src` and observe the broken result.

Choose an image format

Choose between JPEG, PNG, WebP, AVIF, SVG, and video based on the kind of image, transparency, animation, and file size.

The browser can display several image formats. The HTML stays the same:

```
<img src="harbor.webp" alt="Boats in Copenhagen harbor">
```

The format affects file size, quality, and what the image can contain.

Use this as a starting point:

Content	Start with	Why
A photo	WebP, AVIF, or JPEG	Good compression for detailed images
A screenshot	PNG or WebP	Keeps text and sharp edges clear
An image with transparency	PNG or WebP	Supports transparent pixels
A logo or diagram	SVG	Scales without becoming blurry
A long animation	Video	Better controls and compression than GIF

JPEG is still useful for photographs. It does not support transparency.

PNG keeps exact pixel detail and supports transparency. Large photographs saved as PNG are usually much heavier than they need to be.

WebP and AVIF can create smaller files at similar visual quality. Your image editor or build tool can generate them.

SVG is a vector format. The browser draws shapes from instructions instead of displaying a fixed grid of pixels. It works well for logos, icons, and diagrams.

```
<img src="route-map.svg" alt="Cycle route through the city center">
```

SVG is not a good format for a normal photograph.

Compare the result

Export one photo as JPEG, WebP, and AVIF. Open each file at the size it will appear on the page.

Compare two things:

- can you see a meaningful quality difference?
- how large is each file?

Choose the smallest version that still looks good. A fashionable format is not useful if your export looks worse.

Write useful alt text

Decide what an image communicates and provide a concise text alternative, including the correct empty value for decorative images.

Alt text replaces an image for someone who cannot see it. Write what the image contributes in its current context.

```
<img
  src="route-map.png"
  alt="Cycle route from Central Station to the harbor"
>
```

Do not start with “Image of.” A screen reader already announces that it is an image.

Avoid describing details that do not matter. If the page explains a bicycle repair and the photo shows the position of a lever, describe the lever's position. The color of the wall behind it is probably irrelevant.

If the same information is already present in nearby text, keep the alt text short rather than repeating a paragraph.

A purely decorative image should have an empty alt value:

```
<img src="decorative-line.svg" alt="">
```

The empty value tells assistive technology to skip it. Omitting `alt` can cause a screen reader to announce the filename, which is rarely helpful.

If an image contains important text, put that text in the page when possible. Real HTML text is easier to resize, translate, select, and search.

Take this interactive quiz online

Image dimensions and loading

Reserve the right amount of space for an image and defer off-screen images without causing the page to jump while it loads.

Tell the browser an image's intrinsic dimensions:

```
<img
  src="bicycle.jpg"
  alt="A red bicycle parked beside a canal"
  width="1200"
  height="800"
>
```

The values are unitless pixel dimensions of the source image. They give the browser an aspect ratio before the file finishes downloading.

That lets the browser reserve space. Without it, content below the image can jump when the image appears.

CSS can still display the image at another size. The attributes describe the source and reserve the right shape; CSS controls the layout.

Images below the first screen can use native lazy loading:

```
<img
  src="gallery-12.jpg"
  alt="Cyclists crossing the harbor bridge"
  width="1200"
  height="800"
  loading="lazy"
>
```

The browser delays the request until the image is near the viewport. Do not lazy-load the main image visible at the top of the page, because delaying it can make the page feel slower.

Take this interactive quiz online

Figures and captions

Group an image or other self-contained content with a caption that remains connected to it in the document structure.

Use `figure` for self-contained content that is referenced from the surrounding document. Images, diagrams, code samples, and quotations can all be figures.

Add a caption with `figcaption`:

```
<figure>
  <img
    src="harbor-route.png"
    alt="Map showing the harbor cycle route"
    width="900"
    height="600"
```

```
  >
  <figcaption>The harbor route avoids the busiest roads.</figcaption>
</figure>
```

The caption and alt text have different jobs.

The caption is visible to everyone and can add context. The alt text replaces the visual information in the image. Avoid copying the exact same sentence into both.

`figcaption` must be the first or last child of `figure`.

Not every image needs a figure. A small icon inside a button or a logo in the header is not normally a figure. Use it when the content and caption form a meaningful unit.

Responsive images with srcset

Offer several versions of the same image so the browser can choose an appropriate file for the available display width.

A large image looks sharp on a wide screen but wastes bandwidth on a small one. `srcset` lets you offer alternatives:

Diagram: The browser picks an image that fits the available space.

```
<img
  src="harbor-800.jpg"
  srcset="
    harbor-480.jpg 480w,
    harbor-800.jpg 800w,
    harbor-1200.jpg 1200w
  "
  sizes="(max-width: 600px) 100vw, 800px"
  alt="Cyclists crossing the harbor bridge"
  width="1200"
  height="800"
>
```

Each `w` descriptor gives the file's intrinsic width. `480w` means that source is 480 pixels wide.

`sizes` tells the browser how wide the image will be in the layout:

- up to a 600-pixel viewport, it uses the full viewport width
- otherwise, it is about 800 pixels wide

The browser combines that information with screen density and network conditions to choose a source. You do not dictate the exact file.

Keep `src` as a fallback and as the default candidate. Every source should show the same image composition. If the composition needs to change, use `picture` instead.

Take this interactive quiz online

The picture element

Use picture when the image composition or file format should change under specific conditions while keeping a dependable img fallback.

Use `picture` when you need **art direction**: a different crop or composition for a different layout.

```
<picture>
  <source
    media="(max-width: 600px)"
    srcset="harbor-close-up.jpg"
  >
  <img
    src="harbor-wide.jpg"
    alt="Cyclists crossing the harbor bridge"
    width="1200"
    height="700"
  >
</picture>
```

On a small screen, the close crop keeps the cyclists visible. On a wider screen, the fallback `img` shows the full scene.

The browser reads `source` elements from top to bottom and uses the first matching one. The `img` element is required. It provides the fallback, alt text, dimensions, and other image behavior.

`picture` can also offer newer file formats:

```
<picture>
  <source srcset="harbor.avif" type="image/avif">
  <source srcset="harbor.webp" type="image/webp">
  <img src="harbor.jpg" alt="Cyclists beside the harbor">
</picture>
```

If you only need smaller and larger copies of the same composition, `img` with `srcset` is simpler.

Take this interactive quiz online

Audio and video

Embed audio and video with native controls, fallback sources, captions, and behavior that respects the visitor.

Use `audio` to play sound:

```
<audio controls src="interview.mp3">
  <a href="interview.mp3">Download the interview</a>
</audio>
```

The `controls` attribute gives the visitor play, pause, volume, and seeking controls.

Use `video` for video:

```
<video controls width="960" height="540" poster="route-poster.jpg">
  <source src="route.webm" type="video/webm">
  <source src="route.mp4" type="video/mp4">
  <track
    kind="captions"
    src="route-en.vtt"
    srclang="en"
```

```
    label="English"
    default
  >
  <p><a href="route.mp4">Download the route video</a>.</p>
</video>
```

Multiple `source` elements let the browser choose a format it supports. The `track` adds captions for people who cannot hear the audio or prefer to read it.

Avoid autoplaying media with sound. It is disruptive, uses data unexpectedly, and browsers often block it. When autoplay is necessary for a silent visual effect, browsers generally require `muted`, but consider whether a static image would work better.

Take this interactive quiz online

Captions and transcripts

Make audio and video understandable without sound by adding reviewed captions, meaningful sound cues, and a readable transcript.

Media controls let someone operate a video. Captions let them understand its audio.

Add captions with a `track` element:

```
<video controls src="repair.mp4">
  <track
    kind="captions"
    src="repair-en.vtt"
    srclang="en"
    label="English"
    default
  >
</video>
```

The caption file uses WebVTT. A small file looks like this:

```
WEBVTT

00:00:00.000 --> 00:00:03.500
Place the bicycle on a stable surface.

00:00:03.500 --> 00:00:06.000
[bicycle bell rings]
```

Captions include spoken words and meaningful sounds. Identify the speaker when the image does not make that clear.

Do not publish automatic captions without reviewing them. Names, technical words, and punctuation often need correction.

Add a transcript

A transcript presents the same information as normal page text. It helps people who prefer reading, use translation tools, or cannot play the media.

Link it near the media:

```
<p><a href="repair-transcript.html">Read the transcript</a></p>
```

Audio-only content needs a transcript too. If the recording explains something visual, describe that information in the transcript.

Captions and transcripts are not interchangeable. Captions follow the timing of the media. A transcript is easier to scan and search.

The simplest rule is this: do not make sound the only way to receive important information.

Iframes

Embed another page in a controlled frame, give it an accessible title, and limit its capabilities when you do not control the source.

An `iframe` embeds one HTML page inside another:

```
<iframe
  src="https://example.com/map/"
  title="Map of the harbor cycle route"
  width="800"
  height="500"
  loading="lazy"
></iframe>
```

The embedded page has its own document, URL, scripts, and styles. Your page does not absorb its content as normal HTML.

Always provide a useful `title`. Screen-reader users need it to decide whether to enter the embedded document.

Iframes can be expensive because the browser may load an entire application. Lazy-load below-the-fold embeds when appropriate.

If you embed content you do not control, consider the `sandbox` attribute. An empty sandbox applies strong restrictions:

```
<iframe src="/preview/" title="Page preview" sandbox></iframe>
```

You can selectively allow capabilities, but every exception increases what the embedded page can do. Only grant what it needs.

Many sites prevent embedding with HTTP security headers. If an iframe stays blank or reports an error, the remote site may not allow it.

Take this interactive quiz online

Practice: add a project image

Add a local project image with useful alternative text, dimensions, a caption, and a file structure that will keep working online.

Let's add one image to your page and do it properly.

Create an `images` folder inside `my-page`. Put one image there. It can show a project, a drawing, your workspace, or a place you like.

Give the file a short lowercase name, such as `weather-app.webp`.

Add it inside `main`:

```
<figure>
  <img
    src="images/weather-app.webp"
    width="1200"
    height="800"
    alt="Weather dashboard showing the forecast for Copenhagen"
  >
  <figcaption>My first weather dashboard.</figcaption>
</figure>
```

Replace the filename, dimensions, alternative text, and caption with details from your image.

The `src` path is relative to `index.html`. The browser enters the `images` folder and finds the file there.

Check the alternative text

Hide or temporarily rename the image file. Reload the page.

Can the alternative text stand in for the missing image? It should communicate what matters in this context.

Restore the filename when you are done.

If the image is only decoration, use `alt=""`. A project image normally carries information, so it needs useful text.

Check the dimensions

Find the image's real pixel width and height. Put those values in the HTML.

The browser can now reserve the correct shape before the file downloads. This prevents the content below it from jumping.

Done when

The image loads from your local folder, keeps its proportions, has an accurate caption, and still makes sense when it cannot be seen.

Tables and accessibility

Present tabular data and check the page for accessibility.

Build a table

Represent genuinely tabular data with rows, header cells, and data cells instead of using a table as a page layout tool.

Tables are for data with relationships across rows and columns.

```
<table>
  <tr>
```

```
    <th>Route</th>
    <th>Distance</th>
    <th>Difficulty</th>
  </tr>
  <tr>
    <td>Harbor loop</td>
    <td>8 km</td>
    <td>Easy</td>
  </tr>
  <tr>
    <td>Forest trail</td>
    <td>21 km</td>
    <td>Hard</td>
  </tr>
</table>
```

`table` contains the table. `tr` creates a row. Each row contains cells:

- `th` is a header cell
- `td` is a data cell

Use `th` because the cell labels other cells, not because you want bold text.

Do not use tables to arrange a page into columns. That was common in the early Web, but it mixes presentation with data structure and creates a confusing reading order. CSS Grid and Flexbox are made for layout.

Tables can become difficult to use on narrow screens. Keep the data focused and use CSS to handle overflow, but do not remove meaningful columns just to make the table fit.

Take this interactive quiz online

Table structure and captions

Add a caption, group table rows, and associate column or row headers clearly enough for complex data to remain understandable.

A caption gives the table a name. Put it immediately after the opening `table` tag.

Diagram: A caption names the table. Headers describe the data cells.

```
<table>
  <caption>Recommended cycling routes</caption>
  <thead>
    <tr>
      <th scope="col">Route</th>
      <th scope="col">Distance</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th scope="row">Harbor loop</th>
      <td>8 km</td>
    </tr>
```

```
    <tr>
      <th scope="row">Forest trail</th>
      <td>21 km</td>
    </tr>
  </tbody>
</table>
```

`thead` groups header rows. `tbody` groups the main data rows. `tfoot` can group summary rows.

These groups make a long table easier to understand and style. They do not create headers by themselves; you still need `th` elements.

The `scope` attribute makes the relationship explicit:

- `scope="col"` labels cells in the same column
- `scope="row"` labels cells in the same row

For a simple table, browsers can often infer these relationships. Adding scope makes your intent clear and helps assistive technology with more involved tables.

Avoid complicated combinations of `rowspan` and `colspan` when a simpler table would communicate the data better.

Take this interactive quiz online

Spanning cells and complex headers

Use colspan, rowspan, header groups, and explicit header associations without losing the row-and-column relationships in a table.

Some tables need a header that covers several columns.

Use `colspan` to make one cell span them:

```
<table>
  <caption>Workshop registrations</caption>
  <thead>
    <tr>
      <td></td>
      <th id="spring" colspan="2" scope="colgroup">Spring</th>
    </tr>
    <tr>
      <th id="workshop" scope="col">Workshop</th>
      <th id="april" scope="col">April</th>
      <th id="may" scope="col">May</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th id="html" scope="row">HTML</th>
      <td headers="html spring april">18</td>
      <td headers="html spring may">24</td>
    </tr>
  </tbody>
```

```
</table>
```

`colspan="2"` makes the Spring header cover April and May.

`scope="colgroup"` says Spring labels a group of columns. The `headers` attribute makes the complete relationship explicit for each data cell.

The value of `headers` is a space-separated list of `id` values from header cells.

`rowspan` works in the other direction. It makes a cell cover several rows.

Be careful with both attributes. A table that looks obvious as a grid can become difficult to understand when read one cell at a time.

My advice is to keep tables simple. If you need many merged cells and long `headers` values, consider splitting the data into two smaller tables.

Check the relationships

Choose one data cell and answer:

- which row header describes it?
- which column header describes it?
- does a grouped header also apply?

If you cannot answer quickly, the table needs a simpler structure.

Start with native HTML

Use the browser's built-in accessible elements for actions, navigation, and structure before adding custom roles or keyboard behavior.

Accessible HTML usually starts by choosing the element that already has the behavior you need.

For an action, use a button:

```
<button type="button">Save route</button>
```

Do not recreate it with a `div`:

```
<div onclick="saveRoute()">Save route</div>
```

The native button can receive keyboard focus. Enter and Space activate it. Browsers expose its role and disabled state to assistive technology. The `div` gives you none of that for free.

The same principle applies across HTML:

- use `a` with `href` for navigation
- use headings for the document outline
- use `nav` and `main` for landmarks
- use lists for lists
- use form controls for input

ARIA can add accessibility information when native HTML cannot express a custom interface. It does not add browser behavior. A `div role="button"` still needs focus handling, keyboard activation, disabled behavior, and more.

The first rule of ARIA is simple: if a native HTML element already provides the semantics and behavior you need, use it.

Take this interactive quiz online

The accessibility tree

See how semantic HTML becomes a smaller tree of roles, names, values, and states that assistive technology can navigate.

The browser builds more than the DOM.

It also creates an **accessibility tree** for assistive technology. This tree contains the roles, names, values, and states that matter for interaction.

Diagram: Good HTML gives assistive technology useful roles and names.

Consider this button:

```
<button type="button">Save route</button>
```

The accessibility tree exposes a button with the name "Save route." The browser gets both pieces from the native element.

Other examples follow the same pattern:

- `nav` creates a navigation landmark
- `h2` creates a level-two heading
- `img` creates an image whose name comes from `alt`
- `input` gets its name from its associated `label`
- `a` with `href` creates a link

This is why semantic HTML matters even when two elements look identical.

Inspect the tree

Open DevTools and inspect an element. Look for an **Accessibility** pane.

Select a button, link, image, and heading. Check the role and accessible name for each one.

If an image has the wrong name, fix its `alt` text. If a form control has no name, fix its label. Change the HTML source rather than patching the accessibility tree with unnecessary ARIA.

Elements hidden with `display: none` are normally removed from this tree. `aria-hidden="true"` also hides content from assistive technology, but leaves it visible. Do not use it on meaningful or focusable content.

The accessibility tree is a useful debugging view. It shows what your HTML communicates beyond the screen.

A basic accessibility check

Review language, headings, landmarks, link text, image alternatives, and keyboard access before calling an HTML page complete.

Accessibility is part of writing correct HTML. You can catch many problems with a short manual review.

First, check the document:

- does `html` have the correct `lang`?
- is there a descriptive `title`?

- is there one clear main heading?
- do heading levels form a logical outline?
- is the unique content inside `main`?

Then check the content:

- do links describe their destination?
- does every informative image have useful alt text?
- do decorative images have `alt=""`?
- are tables used for data and given proper headers?
- do form controls have labels?

Finally, put the mouse aside. Press Tab and move through the page with the keyboard. You should be able to see where focus is and activate every control.

Automated accessibility tools are useful, but they cannot decide whether alt text communicates the right information or whether a heading describes its section. Use tools after a thoughtful HTML review, not instead of one.

The goal is not to memorize every possible rule. Start with meaningful native elements and test how the page actually works.

Take this interactive quiz online

Practice: add a learning table

Add a small data table to the personal page, then check its caption, headers, landmarks, heading order, and keyboard behavior.

Add a small learning log to your page.

This is tabular data because every entry has the same two fields: a topic and a status.

```
<table>
  <caption>HTML learning log</caption>
  <thead>
    <tr>
      <th scope="col">Topic</th>
      <th scope="col">Status</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th scope="row">Document structure</th>
      <td>Complete</td>
    </tr>
    <tr>
      <th scope="row">Links</th>
      <td>Complete</td>
    </tr>
    <tr>
      <th scope="row">Images</th>
      <td>Practicing</td>
    </tr>
```

```
  </tbody>
</table>
```

Change the rows to reflect what you learned.

Do not use a table to position unrelated content in columns. This table works because each row describes one topic using the same fields.

Run a basic accessibility pass

Now inspect the whole page:

1. Read only the headings. Check their order.
2. Confirm that the page has one `main` element.
3. Check that every informative image has useful alternative text.
4. Press Tab and follow the focus through every link.
5. Zoom the page to 200 percent and make sure nothing important disappears.
6. Inspect the table and confirm its caption and headers are exposed.

Fix one issue at a time. Reload after each change.

Done when

The table makes sense without visual styling and you can use every interactive part of the page with the keyboard.

Forms and debugging

Understand where forms fit and diagnose common HTML errors.

What is a form?

See how the form element collects named values and sends them to a server, while leaving controls and validation to the Forms course.

A form lets a visitor submit information.

Diagram: A form collects name-value pairs and sends them to a server.

```
<form action="/newsletter/subscribe" method="post">
  <label for="email">Email</label>
  <input id="email" name="email" type="email">
  <button>Subscribe</button>
</form>
```

The `form` element groups the controls.

`action` is the URL that receives the submission. `method` is the HTTP method. Forms use GET by default; this example uses POST because it asks the server to change subscription state.

The input's `name` becomes the name sent to the server. Its current value becomes the corresponding value. An input without a name is not included in a normal form submission.

The `label` names the control for everyone and gives it a larger clickable target. Its `for` value matches the input's `id`.

Forms deserve their own course because this small example opens many questions: which control to use, validation, errors, files, `FormData`, JavaScript submission, and server-side safety.

For now, understand where forms fit in an HTML document. We will go much deeper in the Web Forms Course when it is published.

Take this interactive quiz online

What forms send

Follow named form controls into a GET query or POST body and learn which values the browser includes or leaves out.

A form submission is a collection of name-value pairs.

Consider this search form:

```
<form action="/search" method="get">
  <label for="query">Search</label>
  <input id="query" name="q" value="html">
  <button>Search</button>
</form>
```

Submitting it takes the browser to this URL:

```
/search?q=html
```

The name is `q`. The current value is `html`.

The label and `id` make the control understandable and clickable. They do not change the submitted pair. The `name` attribute does that.

With `method="post"`, the browser sends the values in the request body instead of adding them to the URL.

Which controls are included?

The browser normally sends successful named controls:

- a text input sends its current value
- a checked checkbox sends its name and value
- the selected option sends its value
- the button used to submit can send its name and value

An unchecked checkbox is not sent. A disabled control is not sent. A control without `name` is not sent.

This can surprise you when the control is visible in the page but missing on the server.

Use the Network panel to inspect a real submission. Look at the query string for GET or the request payload for POST.

The dedicated Forms course will cover controls, encoding, files, validation, and server-side safety in depth. For HTML, remember the core path:

```
control name + current value -> HTTP request -> server
```

Validate your HTML

Use an HTML validator to find invalid nesting, missing attributes, duplicate IDs, and other mistakes a forgiving browser may silently repair.

Browsers try to render invalid HTML. That is useful for visitors, but it can hide mistakes from you.

An HTML validator checks your source against the standard. Use the Nu HTML Checker and either paste your markup, upload a file, or enter a public URL.

The checker can find problems such as:

- elements nested where they are not allowed
- missing required attributes
- duplicate IDs
- obsolete elements or attributes
- unclosed tags that change the document tree

Read the first error first. One broken tag can cause several later messages, and fixing it may remove the others.

A validator checks syntax and structural rules. It cannot tell whether your heading is useful, your link text is clear, or your alt text describes the important part of an image.

Treat validation as one part of the review:

1. validate the markup
2. inspect the page in the browser
3. use it with the keyboard
4. read the content as a visitor would

Valid HTML is not automatically good HTML, but invalid HTML makes good results harder to trust.

Source code and the DOM

Compare the HTML you wrote with the repaired DOM the browser created, and use the difference to find invalid nesting.

View Page Source shows the HTML the browser received.

The Elements panel shows the DOM the browser built after parsing that HTML. They can be different.

> **Diagram:** The source is what you wrote. The DOM is what the browser built.

For example, a paragraph cannot contain a `div`:

```
<p>
  Introduction
  <div>Important note</div>
</p>
```

The browser closes the paragraph before the `div`. It may also create another empty paragraph for the closing `p` tag.

The page can still look reasonable. The DOM no longer has the structure you intended.

Use valid siblings instead:

```
<p>Introduction</p>
<div>Important note</div>
```

Compare both views

When an element appears in a surprising place:

1. inspect it in the Elements panel
2. look at its parent and nearby siblings
3. open View Page Source
4. compare the same section
5. validate the source

Do not copy the repaired DOM back into your file without understanding it. Fix the invalid source that made the repair necessary.

JavaScript can also change the DOM after the page loads. If the source is valid but the DOM changes later, a script may be responsible.

The distinction is simple:

> Source is the input. The DOM is the browser's result.

Debug HTML with DevTools

Inspect the document tree the browser actually created and trace missing pages or images through the browser's Console and Network panels.

Open the browser's developer tools and select the **Elements** or **Inspector** panel.

This panel shows the DOM: the document tree the browser created after parsing your HTML. It is not always identical to your source. If you nest elements incorrectly, the browser may move or close them while repairing the document.

Use the element picker to select something on the page. Expand its parents and children to understand where it sits in the tree.

If an image does not appear or a link returns a missing page, open the **Network** panel and reload. Find the request and check:

- the requested URL
- the response status
- the response content type

A `404` usually means the path is wrong or the file does not exist. A surprising URL often reveals a relative-path mistake.

The **Console** reports some parsing, loading, and accessibility problems. Read the message, identify the exact element or URL, and fix the source rather than editing the temporary DOM in DevTools.

DevTools edits disappear on refresh. They are excellent for experiments, but the real fix belongs in your HTML file.

Common HTML errors

Diagnose broken nesting, duplicate IDs, missing alternatives, malformed characters, and incorrect paths with a repeatable process.

Most HTML bugs come from a small set of mistakes.

Elements close in the wrong order

Nesting must close from the inside out:

```
<p><strong>Important</strong></p>
```

If the closing order is wrong, the browser has to repair the tree.

An ID appears more than once

An `id` must identify one element in the document.

Duplicate IDs make fragments, labels, and JavaScript selectors unreliable. Search the complete file when the validator reports one.

A relative path starts in the wrong place

This page:

```
/projects/weather/index.html
```

resolves `images/rain.webp` from `/projects/weather/images/rain.webp`.

Inspect the requested URL in the Network panel. It often makes the mistake obvious.

An image has no text alternative

Every `img` needs `alt`. Use useful text for an informative image and `alt=""` for a decorative one.

Text looks like markup

The `<` and `&` characters have special meaning in HTML. Write them as character references when you want the literal character in text:

```
<p>Use &lt;main&gt; for the main content.</p>
<p>Design &amp; development</p>
```

Debug in a fixed order

When something breaks:

1. validate the HTML
2. inspect the DOM around the problem
3. check failed requests in Network
4. read Console messages
5. reduce the markup to the smallest failing example

Change one thing, reload, and check again. Random edits make the original problem harder to see.

Practice: finish your page

Validate and debug the complete personal page, fix broken paths and nesting, and perform a final browser and keyboard review.

Your page now contains a document shell, text, links, landmarks, an image, and a table.

Let's finish it with the same checks you would use on a real site.

Validate the HTML

Run `index.html` and `about.html` through the HTML validator.

Fix errors first. Then review warnings and decide whether they apply.

Common mistakes include:

- an element closed in the wrong order
- a duplicate `id`
- an image without `alt`
- a heading or table element in an invalid place

Validate again after your changes.

Break one thing on purpose

Change the image path so it points to a filename that does not exist. Reload the page and inspect the Network panel.

Find the failed request. Look at its URL and status.

Restore the correct path and confirm the request succeeds. This small exercise makes broken paths much easier to diagnose later.

Inspect the document

Open the Elements panel and expand `html`, `head`, and `body`.

Compare the DOM with the source you wrote. If the browser moved or inserted an element, inspect the surrounding markup for invalid nesting.

Do the final pass

- open every page and click every link
- use Tab to reach every interactive element
- zoom to 200 percent
- reload with the Network panel open
- view the page source
- read the heading outline
- check the page at a narrow browser width

Do not aim for a beautiful page yet. CSS will handle presentation.

The goal is a page with clear content and reliable structure.

Done when

Both pages validate, every local file loads, every link works, and you can explain why each major element is there.

Final HTML check

Review the complete course with one practical page checklist and a mixed quiz covering documents, text, links, media, tables, and accessibility.

You now have enough HTML to build a solid document without hiding behind a framework.

Before you finish a page, review it from the outside in.

Document

- start with `<!doctype html>`
- set the document language
- include UTF-8, viewport metadata, and a useful title
- keep visible content in `body`

Content

- use one clear main heading and logical subheadings
- use paragraphs and lists for the content they represent
- write descriptive link text and check relative paths
- choose semantic containers before generic ones

Media and data

- write contextual alt text
- include image dimensions
- add captions and media controls where needed
- use tables only for tabular data, with headers and captions

Accessibility and debugging

- prefer native elements
- navigate the page with a keyboard
- validate the HTML
- inspect the browser's DOM and failed requests

If a page looks wrong, resist the urge to add more markup immediately. Check the structure, nesting, and paths first. Simple HTML is easier to understand and repair.

Complete these four short checks to finish the course.

Take this interactive quiz online

Take this interactive quiz online

Take this interactive quiz online

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/html/.