Web System Development

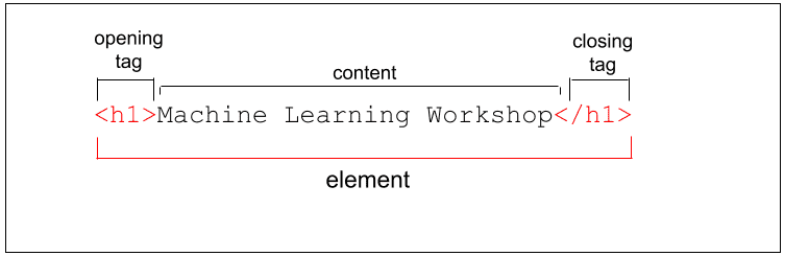
Module 01: HTML

Overview of HTML

- HyperText Markup Language, or HTML, is the standard **markup language** for describing the structure of documents displayed on the web.
- HTML consists of a series of elements and attributes which are used to mark up all the components of a document to structure it in a meaningful way.
- HTML documents are basically a **tree of nodes** , including HTML elements and text nodes.
  - HTML elements provide the semantics and formatting for documents, including creating paragraphs, lists and tables, and embedding images and form controls.
  - Each element may have multiple attributes specified.
  - Many elements can have content, including other elements and text. Other elements are empty, with the tag and attributes defining their function.

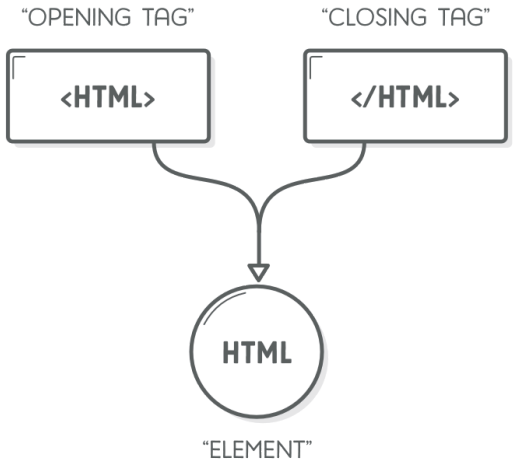
HTML Elements

- HTML elements are delineated by **tags** , written using angle brackets (`<` and `>`).
- Our page title is a heading, level one, for which we use the `<h1>` tag.
  - The actual title, "Machine Learning Workshop", is the content of our element.
  - The content goes between the open and closing tags.
  - The entire thing—the opening tag, closing tag, and the content—is the element.
  - The closing tag is the same tag as the opening tag, preceded by a slash.
- Browsers do not display the tags. The tags are used to interpret the content of the page.



Structure of a web page

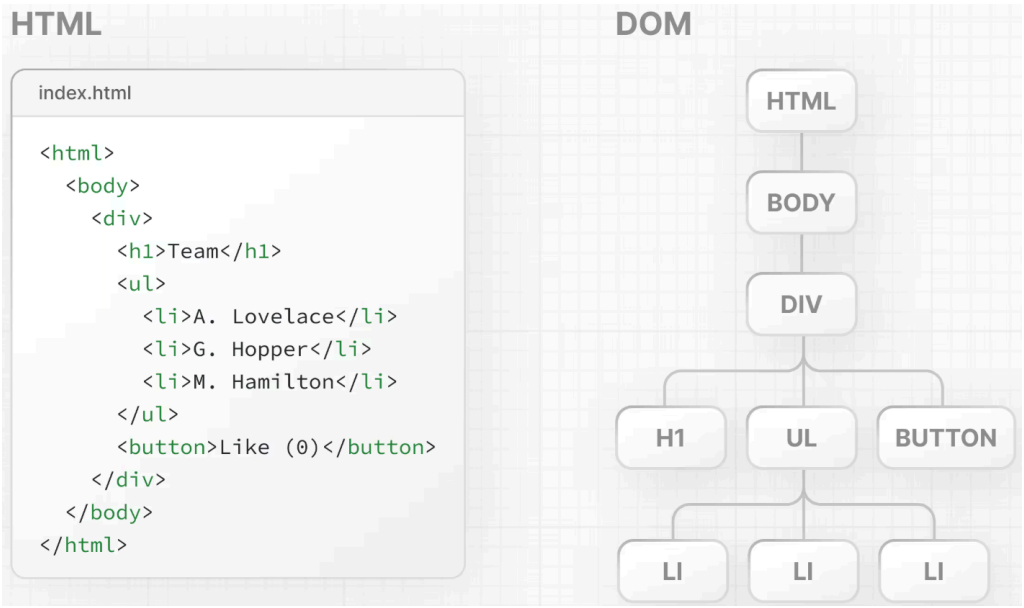
```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <!-- Metadata goes here -->
5   </head>
6   <body>
7     <!-- Content goes here -->
8   </body>
9 </html>
```



- First, we need to tell browsers that this is an **HTML5** web page with the `<!doctype html>` line. This is just a special string that browsers look for when they try to display our web page, and it always needs to look exactly like it does above.
- Then, our entire web page needs to be wrapped in `<html>` tags.
  - The actual `<html>` text is called an "opening tag", while `</html>` is called a "closing tag".
  - Everything inside of these tags are considered part of the `<html>` "element", which is this ethereal thing that gets created when a web browser parses your HTML tags.

HTML DOM

When a user visits a web page, the server returns an HTML file to the browser that may look like this:

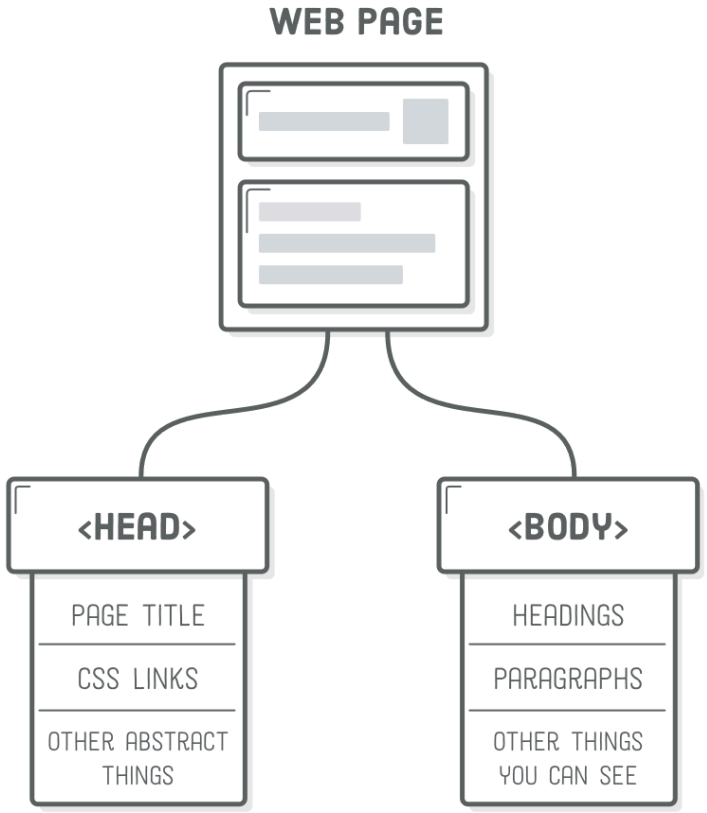


The browser then reads the HTML and constructs the Document Object Model (DOM).

Head and Body

Inside of the `<html>` element, we have two more elements called `<head>` and `<body>` . A web page's head contains all of its metadata, like the page title, any CSS stylesheets, and other things that are required to render the page but you don't necessarily want the user to see.

The bulk of our HTML markup will live in the `<body>` element, which represents the visible content of the page. Note that opening up our page in a web browser wont display anything, since it has an empty `<body>` .



The purpose of this `<head>` / `<body>` split will become clearer in a few chapters after we start working with CSS.

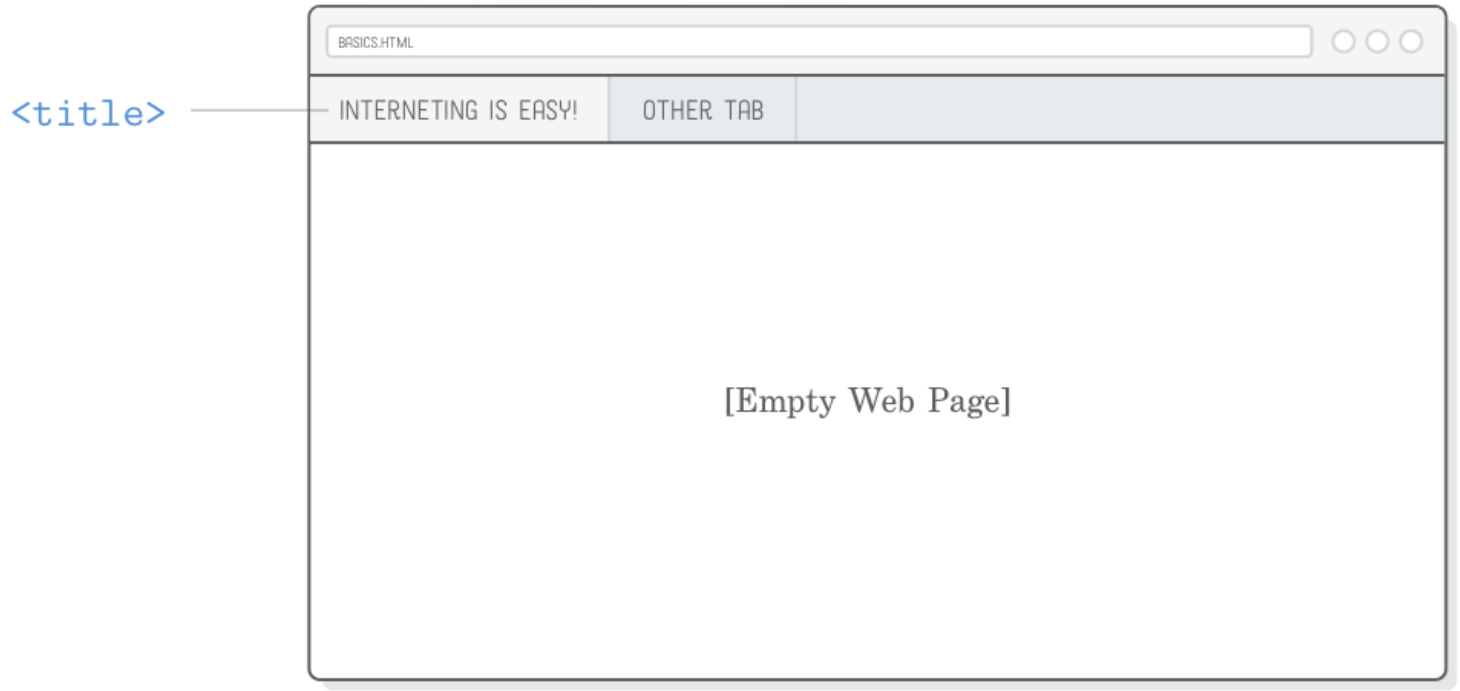
HTML Comments

- Notice the HTML comment syntax in the above snippet. Anything that starts with `<!--` and ends with `-->` will be completely ignored by the browser.
- HTML comments are used to leave notes in your code that won't be seen by the browser.
- They're useful for documenting your code and making notes to yourself or temporarily disabling certain elements.

Page Titles

One of the most important pieces of metadata is the title of your web page, defined by the aptly named `<title>` element. Browsers display this in the tab for your page, and Google displays it in search engine results.

example01.html



Nested Elements

- Notice how all the HTML tags in our web page are neatly **nested** .
- HTML tags should be **closed in the reverse order** of which they were opened.
- It's very important to ensure that there are no overlapping elements.
- For instance, the `<title>` element is supposed to be inside of the `<head>` , so you'd never want to add the closing `</head>` tag before the closing `</title>` tag.

```
<!-- (Don't ever do this) -->
<head>
  <title>Pay attention to the tags!</title>
</title>
```

Text Basics

HTML provides a number of elements that let you mark up text.

- `<p>` for paragraphs
- `<h1>` for headings

Paragraphs

Titles are all well and good, but let's do something we can actually see. The `<p>` element marks all the text inside it as a distinct paragraph. Try adding the following `<p>` element to the body of our web page:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My First Web Page</title>
  </head>
  <body>
    <p>First, we need to learn some basic HTML.</p>
  </body>
</html>
```

Also note how the `<p>` and `<title>` elements are indented twice, while `<body>` and `<head>` are indented once.

Indenting nested elements like this is an important best practice that makes your HTML easier to read for other developers (or for yourself if you come back 5 months from now and want to change some stuff).

Headings

- Headings are the primary way you mark up different sections of your content. They define the **outline** of your web page as **both humans and search engines see it** , which makes selecting relevant headings essential for a high-quality web page.
- Headings are like titles, but they’re actually displayed on the page.
- HTML provides six levels of headings, and the corresponding elements are: `<h1>` , `<h2>` , `<h3>` , … , `<h6>` .
- The higher the number, the less prominent the heading.
- By default, browsers render less important headings in smaller fonts.

```
<h1>Main page heading</h1>
<p>First, we need to learn some basic HTML.</p>

<h2>Headings</h2>
<p>Headings define the outline of your site. There are six levels of
headings.</p>
```

HTML Entities

- An *HTML entity* is a special character that can't be represented as plain text in an HTML document.
- This typically either means it's a **reserved character** in HTML or you don't have a key on your keyboard for it.

Reserved Characters

- The `<` , `>` , and `&` characters are called reserved characters because they aren't allowed to be inserted into an HTML document without being encoded.
- This is because they mean something in the HTML syntax:
  - `<` begins a new tag,
  - `>` ends a tag, and, as we're about to learn,
  - `&` sets off an HTML entity.
- Entities always begin with an ampersand (`&`) and end with a semicolon (`;`). In between, you put a special code that your browser will interpret as a symbol. In this case, it interprets `lt` , `gt` , and `amp` as less-than, greater-than, and ampersand symbols, respectively.

```
<h2>HTML Entities</h2>
<p>
  There are three reserved characters in HTML: &lt; &gt; and &amp;.
  You should always use HTML entities for these three characters.
</p>
```

Lists

- HTML provides us with a few different ways to mark up lists. There are:
  - unordered lists (`<ul>`),
  - ordered lists (`<ol>`),
  - and description lists (`<dl>`).
- List items (`<li>`) are nested within ordered lists and unordered lists.
- Inside a description list, you'll find description terms (`<dt>`) and description details `<dd>.`

Unordered Lists

- The `<ul>` element is the parent element for unordered lists of items.
- The only children of a `<ul>` are one or more `<li>` list item elements.
- By default, each unordered list item is prefixed with a bullet.
- You need to close out your lists with a `</ul>` .

```
<h2>Lists</h2>

<p>This is how you make an unordered list:</p>

<ul>
  <li>Add a "ul" element (it stands for unordered list)</li>
  <li>Add each item in its own "li" element</li>
  <li>They don't need to be in any particular order</li>
</ul>
```

Nested Elements in Lists

The HTML specification defines strict rules about what elements can go inside other elements. In this case, `<ul>` elements should only contain `<li>` elements, which means you should never ever write something like this:

```
<!-- (This is bad!) -->
<ul>
  <p>Add a "ul" element (it stands for unordered list)</p>
</ul>
```

Instead, you should wrap that paragraph with `<li>` tags:

```
<!-- (Do this instead) -->
<ul>
  <li><p>Add a "ul" element (it stands for unordered list)</p></li>
</ul>
```

How do we know about these strict rules?

- How do we know that `<ul>` only accepts `<li>` elements and that `<li>` allows nested paragraphs?
- Because the Mozilla Developer Network (MDN) says so. MDN is a superb HTML element reference.
- Whenever you're not sure about a particular element, do a quick Google search for **“MDN <some-element>”** .

Ordered Lists

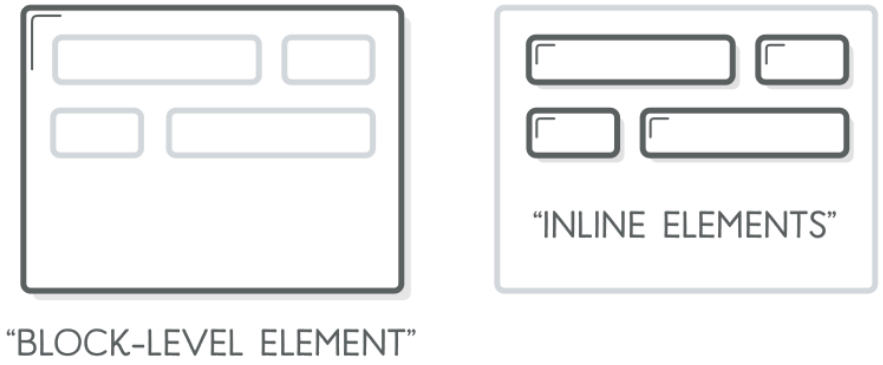
- With an unordered list, rearranging the `<li>` elements shouldn't change the meaning of the list. .
- If the sequence of list items does matter, you should use an "ordered list" instead.
- To create an ordered list, simply change the parent `<ul>` element to `<ol>`

```
<p>This is what an ordered list looks like:</p>

<ol>
  <li>Notice the new "ol" element wrapping everything</li>
  <li>But, the list item elements are the same</li>
  <li>Also note how the numbers increment on their own</li>
  <li>You should be noticing things is this precise order, because this is
      an ordered list</li>
</ol>
```

Block-Level vs. Inline Elements

- So far, we've only been working with *block-level* elements.
- The other major type of content is *inline elements* which are treated a little bit differently.
- Block-level elements are always drawn on a new line, while inline elements can affect sections of text anywhere within a line.



Emphasis (Italic) Elements

For instance, `<p>` is a block-level element, while `<em>` is an inline element that affects a span of text *inside* of a paragraph. It stands for “emphasis”, and it's typically displayed as italicized text.

```
<h2>Inline Elements</h2>

<p><em>Sometimes</em>, you need to draw attention to a particular word or
phrase.</p>

<!-- It's important that you properly nest your HTML elements. -->
<!-- (Again, don't ever do this) -->
<p>This is some <em>emphasized text</em></p>
```

Strong (Bold) Elements

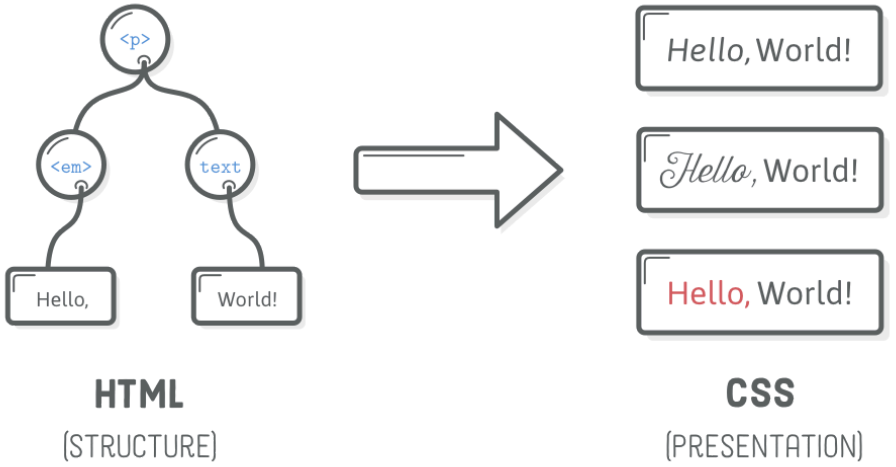
If you want to be more emphatic than an `<em>` tag, you can use `<strong>` . It's an inline element just like `<em>` , and looks like this:

```
<p>Other times you need to <strong>strong</strong>ly emphasize the importance
  of a word or phrase.</p>

<p><em><strong>And sometimes you need to shout!</strong></em></p>
```

Structure vs Presentation

- You might be wondering why we're using the terms *emphasis* and *strong* instead of *italic* and *bold* .
- That brings us to an important distinction between HTML and CSS.
- HTML markup should provide **semantic** information about your content—not **presentational** information.
- In other words, HTML should define the structure of your document, leaving its appearance to CSS.



Why do we use `<em>` and `<strong>` ?

- The pseudo-obsolete `<b>` and `<i>` elements are classic examples of this.
- They used to stand for *bold* and *italic* , respectively, but HTML5 attempted to create a clear separation between a document's structure and its presentation.
- Thus, `<i>` was replaced with `<em>` , since emphasized text can be displayed in all sorts of ways aside from being italicized (e.g., in a different font, a different color, or a bigger size). Same for `<b>` and `<strong>` .

Empty HTML Elements

- The HTML tags we've encountered so far either wrap text content (e.g., `<p>`) or other HTML elements (e.g., `<ol>`).
- That's not the case for all HTML elements. Some of them can be *empty* or *self-closing* .
- Line breaks and horizontal rules are the most common empty elements you'll find.

Line Breaks

HTML condenses consecutive spaces, tabs, or newlines (together known as *whitespace*) into a single space.

```
<h2>Empty Elements</h2>

<p>Thanks for reading! HTML should be getting easier now.</p>

<p>Regards,
The Authors</p>
```

The newline after `Regards` in the above snippet will be transformed into a space instead of displaying as a line break

Add a Line Break

- This behavior may seem counter intuitive, but web developers often set their text editor to limit line length to around 80 characters.
- As a programmer, it's easier to manage code this way, but having each of the newlines show up in the rendered page would severely mess up the intended page layout.
- To tell the browser that we want a **hard line break** , we need to use an explicit `<br/>` element.

```
<p>Regards,<br>
  The Authors</p>
```

Be Careful with Line Breaks

However, be very careful not to abuse the `<br/>` tag. Each one you use should still convey *meaning* —you shouldn't use it to, say, add a bunch of space between paragraphs:

```
<!-- You shouldn't do this -->
<p>This paragraph needs some space below it...</p>
<br><br><br><br><br><br><br><br>
<p>So, I added some hard line breaks.</p>
```

This kind of presentational information should be defined in your **CSS** instead of your **HTML** .

Horizontal Rules

- The `<hr/>` element is a “horizontal rule”, which represents a thematic break.
- The transition from one scene of a story into the next or between the end of a letter and a postscript are good examples of when a horizontal rule may be appropriate.

```
<h2>Empty Elements</h2>

<p>Thanks for reading! HTML should be getting easier now.</p>

<p>Regards,<br>
The Authors</p>

<hr>

<p>P.S. This page might look like crap, but we'll fix that with some CSS
soon.</p>
```

- One of the themes for this chapter has been the separation of content (HTML) from presentation (CSS), and `<hr/>` is no different.
- Like `<em>` and `<strong>` , it has a default appearance (a horizontal line), but once we start working with CSS, we'll be able to render it as more space between sections, a decorative accent character, or pretty much anything else we want.
- Like `<br/>` , `<hr/>` should carry **meaning** . Don't use it when you just want to display a line for the sake of aesthetics. For that, you'll want to use the CSS `border` property, which we'll discuss in a few chapters.

Optional Trailing Slash

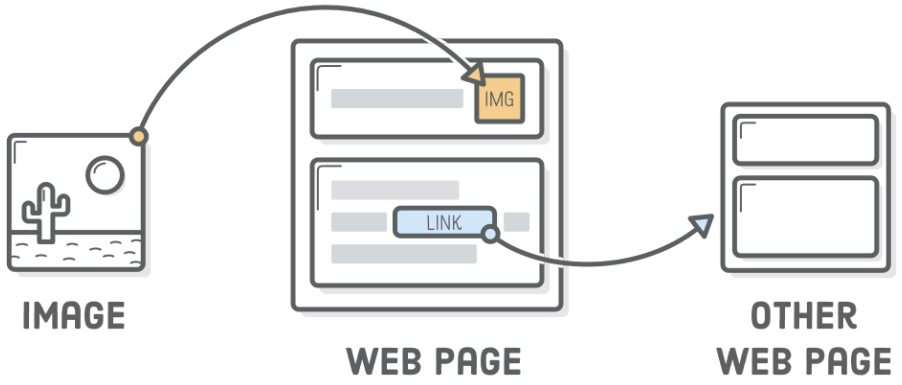
- The trailing slash (`/`) in all empty HTML elements is entirely optional. The above snippet could also be marked up like this (note the lack of `/` in the `<br>` and `<hr>` tags)
- It doesn't really make a difference which convention you choose, but pick one and stick to it for the sake of consistency.

```
<p>Regards,<br>The Authors</p>

<hr>
```

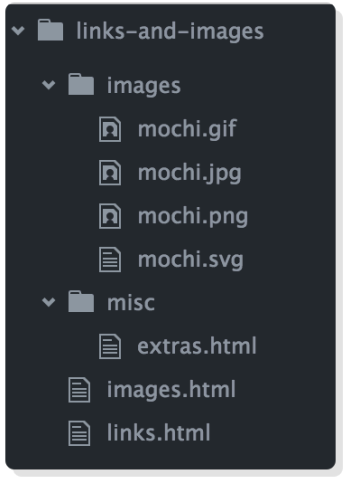
Links and Images

- Links and images are fundamentally different from those elements in that they deal with *external* resources.
- Links point the user to a different HTML document, and images pull another resource into the page.
- As we start working with multiple files, we'll discover the importance of being an **organized** web developer.



Setup

We'll be working with **three separate web pages** this chapter, along with **a few image files** of various formats:



- Create a new folder called *links-and-images* to store all our files.
- Add two new files to that folder called *links.html* and *images.html* and insert the HTML templates below.
- Our last page will help us demonstrate relative links. Create a new folder in *links-and-images* called *misc* , then add a new file called *extras.html*

```
<!-- /links.html -->

<!DOCTYPE html>
<html>
  <head>
    <title>Links</title>
  </head>
  <body>
    <h1>Links</h1>
  </body>
</html>
```

```
<!-- /images.html -->

<!DOCTYPE html>
<html>
  <head>
    <title>Images</title>
  </head>
  <body>
    <h1>Images</h1>
  </body>
</html>
```

```
<!-- /misc/extras.html -->

<!DOCTYPE html>
<html>
  <head>
    <title>Extras</title>
  </head>
  <body>
    <h1>Extras</h1>
  </body>
</html>
```

Image Downloads

- We'll be embedding images in our `images.html` file.
- Here, you can download these example mochi images .
- Unzip them in your `links-and-images` folder, keeping the parent `images` folder from the ZIP file.

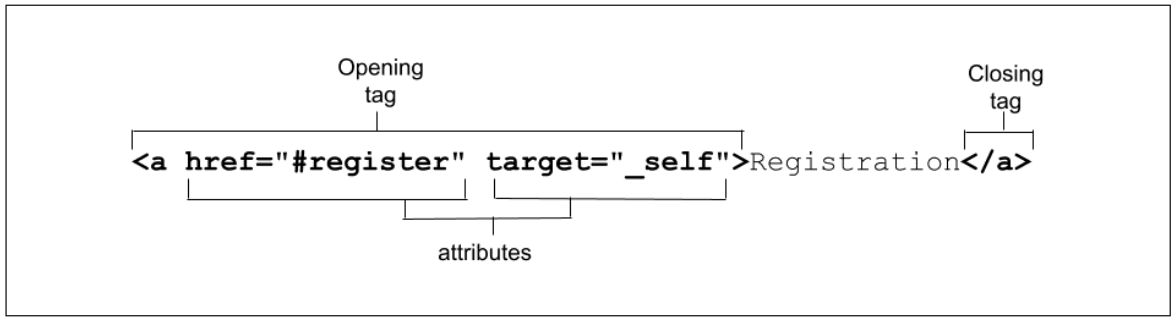
Anchors

- Links are created with the `<a>` element, which stands for *anchor* .
- It works just like all the elements in the previous chapter: when you wrap some text in `<a>` tags, it alters the meaning of that content.
- Let's take a look by adding the following paragraph to the `<body>` element of `links.html` :

```
<p>This example is about links and <a>images</a>.</p>
```

- However, if you load the page in a web browser, you'll notice that the `<a>` element doesn't look like a link at all.
- Yes, unfortunately, the `<a>` element on its own doesn't do much of anything.

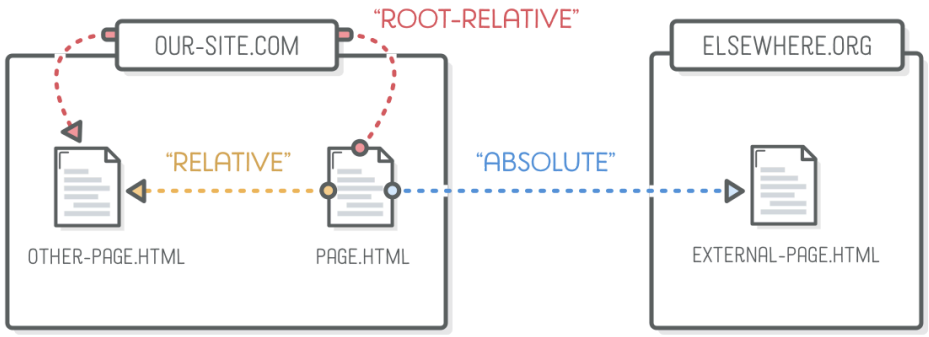
Links



- In the same way that an element adds meaning to the content it contains, an HTML **attribute** adds meaning to the element it's attached to.
- Different elements allow different attributes, and you can refer to MDN for details about which elements accept which attributes.
- Right now, we're concerned with the `href` attribute because it determines where the user goes when they click an `<a>` element.
- Notice how attributes live inside the opening tag. The attribute **name** comes first, then an equal sign, then the **value** of the attribute in either single or double quotation marks.

```
<p>This example is about links and <a href="images.html">images</a>.</p>
```

Absolute, Relative and Root-Relative Links



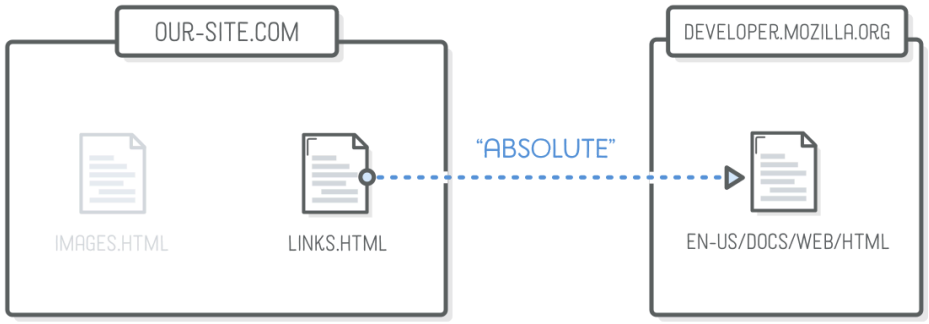
- A website is just a collection of **HTML files** organized into **folders** .
- To refer to those files from inside of another file, the Internet uses Uniform Resource Locators (*URLs*).
- Depending on what you're referring to, URLs can take different forms.

```
<!-- Add the following content to our links.html file -->

<p>This particular page is about links! There are three kinds of links:</p>

<ul>
  <!-- Add <li> elements here -->
</ul>
```

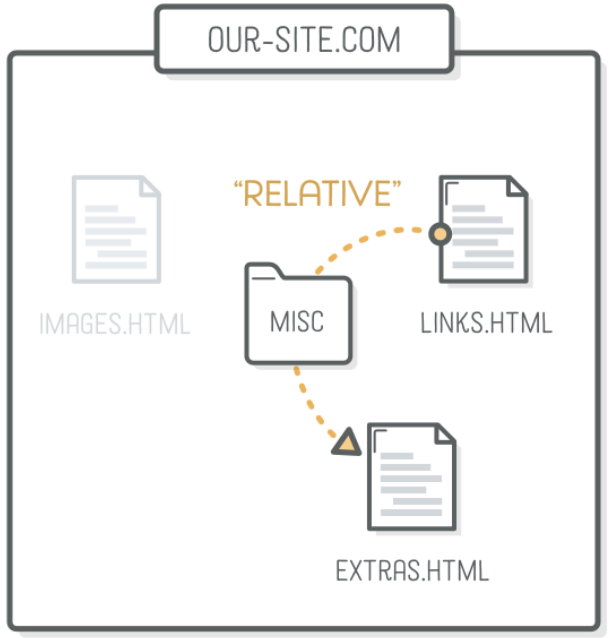
Absolute Links

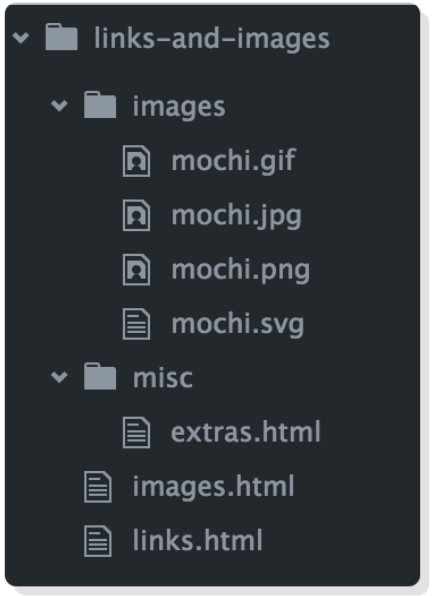


Absolute links are the most detailed way you can refer to a web resource. They start with the **scheme** (typically `http://` or `https://`), followed by the **domain name** of the website, then the **path of the target web page** .


```
<li>Absolute links, like to
  <a href="https://developer.mozilla.org/en-US/docs/Web/HTML">Mozilla
  Developer Network</a>, which is a very good resource for web
  developers.</li>
```

Relative Links





- Relative links point to **another file** in your **website** from the vantage point of the file you're editing.
- It's implied that the **scheme** and **domain name** are the **same** as the current page, so the only thing you need to supply is the **path** .

```
<!-- Here's how we can link to our extras.html file from inside of links.html: -->

<li>Relative links, like to our <a href="misc/extras.html">extras page</a>.</li>
```

Parent Folders

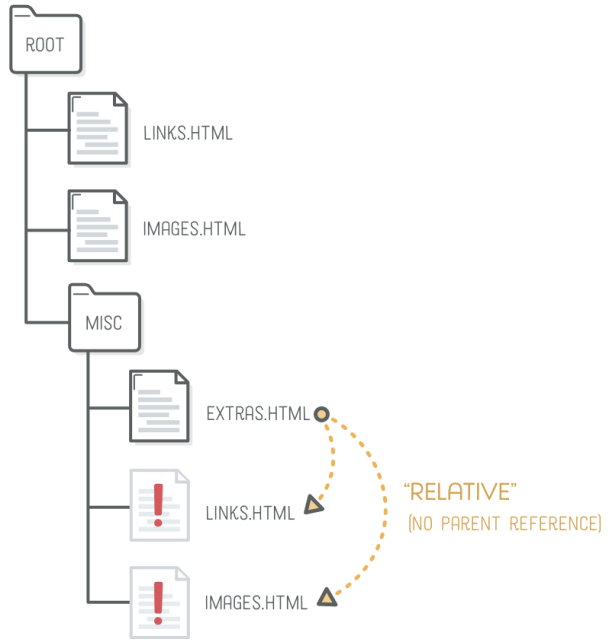
- That works for referring to files that are in the same folder or a deeper folder.
- What about linking to pages that are in a directory *above* the current file?
- Let's try creating relative links to `links.html` and `images.html` from within our `extras.html` file.

```
<!-- Add this to extras.html -->

<p>
  This page is about miscellaneous HTML things, but you may also be interested in <a href='links.html'>links</a> or
  <a href='images.html'>images</a>.
</p>
```

Parent Folders

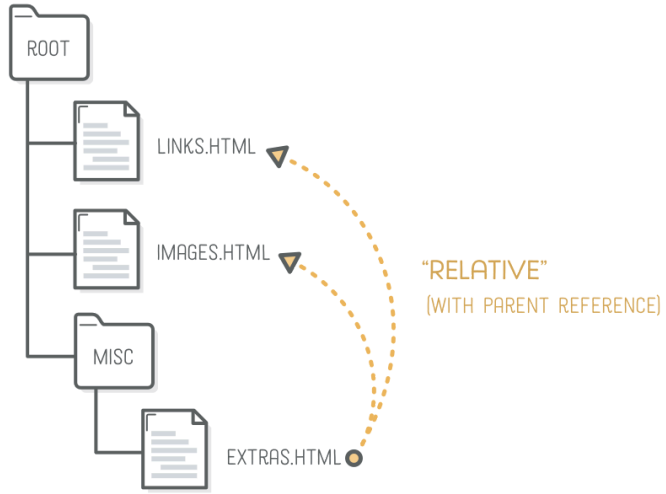
- When you click either of those links in a web browser, it will complain that the page doesn't exist.
- Examining the address bar, you'll discover that the browser is trying to load `misc/links.html` and `misc/images.html` —it's looking in the wrong folder!
- That's because our links are *relative* to the location of `extras.html` , which lives in the `misc` folder.



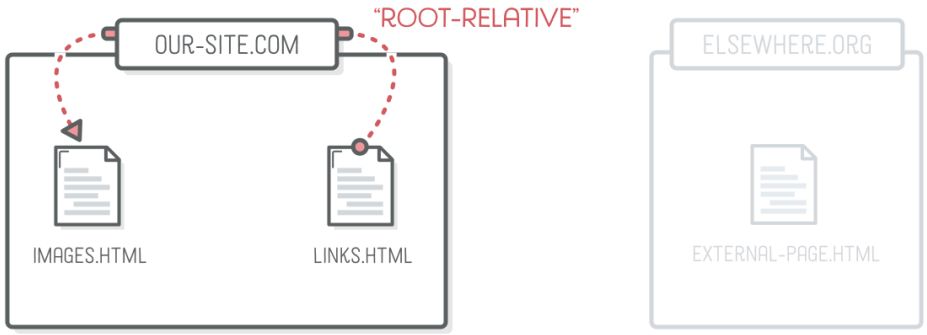
How to fix this?

```
<p>
  This page is about miscellaneous HTML things, but you may also be
  interested in <a href="../links.html">links</a> or <a href="../images.html">images</a>.
</p>
```

- To fix this, we need the `..` syntax.
- Two consecutive dots in a file path represent a pointer to the **parent** directory
- This is like saying, “I know `extras.html` is in the `misc` folder. Go up a folder and look for `links.html` and `images.html` in there.”



Root-Relative Links



- Root-relative links are similar to the previous section, but instead of being relative to the current page, they're relative to the **root** of the entire website.
- For instance, if your website is hosted on `our-site.com` , all root-relative URLs will be relative to `our-site.com` .
- Unfortunately, there is one caveat to our discussion of root-relative links: we are using local HTML files instead of a website hosted on a web server. This means we **won't be able to experiment with root-relative links** . But, if we did have a real server, the link to our home page would look like this:

```
<!-- This won't work for our local HTML files -->
<!-- The only difference between a root-relative link and a relative one is that the former starts with a forward slash.  -->
<li>Root-relative links, like to the <a href='/'>home page</a> of our website, but those aren't useful to us right now.</li>
```

Link Targets

- Attributes alter the meaning of HTML elements, and sometimes you need to modify more than one aspect of an element.
- For example, `<a>` elements also accept a `target` attribute that defines where to display the page when the user clicks the link.
- By default, most browsers replace the current page with the new one. We can use the `target` attribute to ask the browser to open a link in a new window/tab.
- The `target` attribute has a few pre-defined values that carry special meaning for web browsers, but the most common one is `_blank` , which specifies a new tab or window. You can read about the rest on MDN .

```
<li>
  Absolute links, like to
  <a href="https://developer.mozilla.org/en-US/docs/Web/HTML" target="_blank">Mozilla Developer Network</a>, which is a very good
     resource for web developers.
</li>
```

Naming Conventions

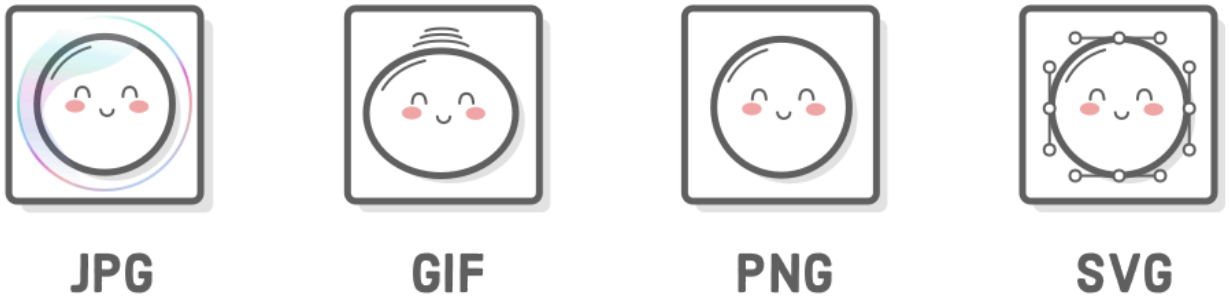
- You'll notice that none of our files or folders have *spaces* in their names. That's on purpose.
- Spaces in URLs require special handling and should be avoided at all costs.
- To see what we're talking about, try creating a new file in our `links-and-images` project called `spaces are bad.html`
- Add a little bit of text to it, then open it in Google Chrome or Safari (Firefox cheats and preserves the spaces).
- In the address bar, you'll see that all our spaces have been replaced with `%20` : `links-and-images/spaces%20are%20bad.html`
- Spaces aren't allowed in URLs, and that's the special encoding used to represent them.
- Instead of a space, you should always use a **hyphen** , as we've been doing throughout this tutorial.
- It's also a good idea to use all lowercase characters for consistency.
- Notice how there's a direct connection between our **file/folder names and the URL** for the web page they represent.
  - The names of our folders and files determine the slugs for our web pages.
  - They're visible to the user, which means you should put in as much effort into naming your files as you put into creating the content they contain.

Images

- Images are included in web pages with the `<img/>` tag and its `src` attribute, which points to the image file you want to display.
- Notice how it's an empty element like `<br/>` and `<hr/>` .
- Retina displays and mobile devices make image handling a little bit more complicated than a plain old `<img/>` tag. We'll deal with this later in the course.
- Also be sure to check out the `<figure>` and `<figcaption>` element in the Semantic HTML chapter.
- For now, let's focus on the many image formats floating around the Internet.

Image Formats

There's four main image formats in use on the web, and they were all designed to do different things.



Understanding their intended purpose goes a long way towards improving the quality of your web pages.

JPG Images

- JPG images are designed for handling **large color palettes** without exorbitantly increasing **file size** .
- This makes them great for **photos** and images with lots of **gradients** in them.
- On the other hand, JPGs don't allow for **transparent** pixels, which you can see in the white edges of the image below if you look real close

```
<!-- links-and-images/images.html -->
<p>This page covers common image formats.</p>
<h2>JPGs</h2>
<img src="images/mochi.jpg">
```

GIF Images

- GIFs are the go-to option for simple **animations** , but the trade off is that they're somewhat **limited in terms of color palette** —never use them for photos.
- Transparent pixels are a binary option for GIFs, meaning you can't have **semi-opaque** pixels. This can make it difficult to get high levels of detail on a transparent background.
- For this reason, it's usually better to use **PNG** images if you don't need animation.

```
<!-- links-and-images/images.html -->
<h2>GIFs</h2>
<p>GIFs are good for animations.</p>
<img src="images/mochi.gif">
```

PNG Images

- PNGs are great for **anything** that's **not a photo or animated** .
- For **photos** , a PNG file of the same quality (as perceived the human eye) would generally be bigger than an equivalent **JPG** file.
- However, they do deal with **opacity** just fine, and they don't have color palette limitations. This makes them an excellent fit for icons, technical diagrams, logos, etc.

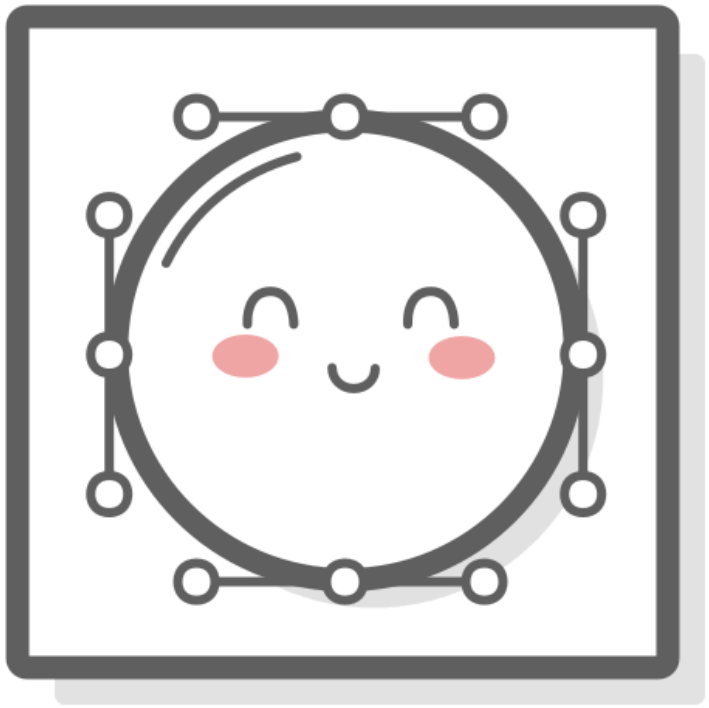
```
<!-- links-and-images/images.html -->
<h2>PNGs</h2>
<p>PNGs are good for diagrams and icons.</p>
<img src="images/mochi.png">
```

SVG Images

- Unlike the pixel-based image formats above, SVG is a **vector-based** graphics format, meaning it can scale up or down to *any* dimension without loss of quality.
- This property makes SVG images a wonderful tool for **responsive design** . They're good for pretty much all the same use cases as PNGs, and **you should use them whenever you can** .

SVG Images

- These 300x300 pixel images were originally 100x100 pixels, but scaling them up clearly shows the difference between them.
- Notice the crisp, clean lines on the left SVG image, while the PNG image on the right is now very pixelated.
- Despite being a vector format, SVGs can be used exactly like their raster counterparts.

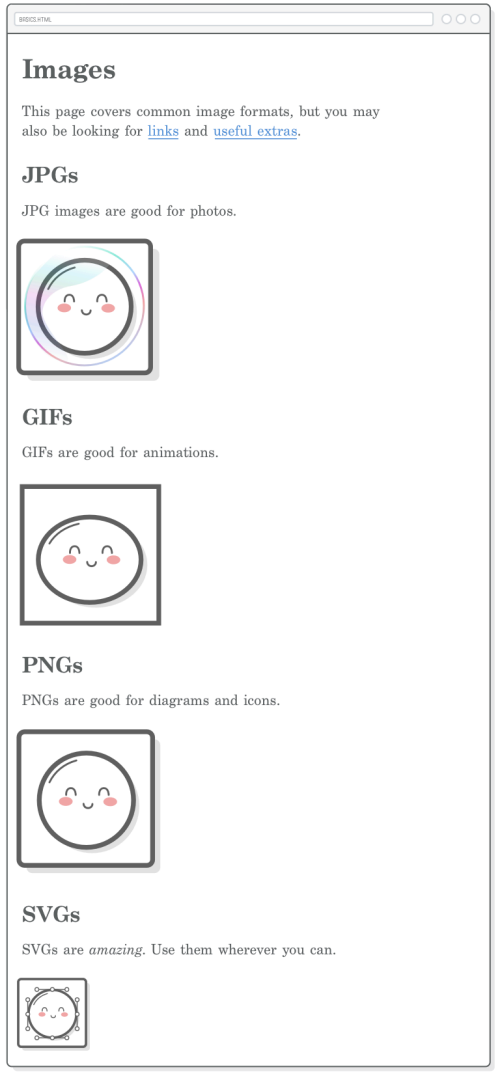


Caveats

- There is one potential issue with SVGs: for them to display consistently across browsers, you need to convert any text fields to outlines using your image editor (e.g., Sketch).
- If your images contain a lot of text, this can have a big impact on file size.

Image Dimensions

- By default, the `<img/>` element uses the **inherit dimensions of its image file** .
- Our JPG, GIF, and PNG images are actually 150x150 pixels, while our SVG mochi is only 75x75 pixels.
- To get our pixel-based images down to the intended size (75x75), we can use the `<img/>` element's `width` attribute.
- The `width` attribute sets an explicit dimension for the image.
- There's a corresponding `height` attribute, as well.
- Setting only one of them will cause the image to scale proportionally, while defining both will stretch the image.
- Dimension values are specified in pixels, and you should never include a unit (e.g., `width='75px'` would be incorrect).




```
<img src="images/mochi.jpg" width="75">
```

Image Alt Text

- Adding `alt` attributes to your `<img/>` elements is a best practice.
- It defines a **text alternative** to the image being displayed.
- This has an impact on both search engines and users with text-only browsers (e.g., people that use *text-to-speech* software due to a vision impairment).
- For more examples of how to use the `alt` attribute, please refer to the ridiculously detailed official specification .

```
<img src="images/mochi.jpg" width="75" alt="A mochi ball in a bubble">
```

Tables

- HTML tables are used for displaying tabular data with rows and columns.
- The decision to use a `<table>` should be based on the content you are presenting and your users' needs in relation to that content.
- If data is being presented, compared, sorted, calculated, or cross-referenced, then `<table>` is probably the right choice.
- If you simply want to lay out non-tabular content neatly, such as a large group of thumbnail images, tables are not appropriate: instead, create a list of images .

Table Elements

- The <table> tag wraps the table content, including all the table elements.
- Inside the `<table>` , you'll find
  - the table headers (`<thead>`),
  - table bodies (`<tbody>`),
  - and, optionally, table footers (`<tfoot>`).
- Each of these is made up of table rows (`<tr>`).
  - Rows contain table header (`<th>`) and table data (`<td>`) cells which, in turn, contain all the data.
  - In the DOM, before any of this, you may find two additional features: the table caption (`<caption>`) and column groups (`<colgroup>`).
  - Depending on whether or not the `<colgroup>` has a `span` attribute, it may contain nested table column (`<col>`) elements.

The table's children in order

1. <caption> element
2. <colgroup> elements
3. <thead> elements
4. <tbody> elements
5. <tfoot> elements

Table Caption

- The `<caption>` provides a descriptive, programmatically associated table title. It is visible and available to all users by default.
- It should be the first element nested in the `<table>` element. Including it lets all users know the purpose of the table immediately without having to read the surrounding text.
- The caption appears outside the table.

```
<table>
  <caption>A summary of the planets of our solar system</caption>
</table>
```

Data Sectioning

The content of tables is made up of up to three sections:

- zero or more table headers (`<thead>`) ,
- table bodies (`<tbody>`),
- and table footers (`<tfoot>`).

All are optional, with zero or more of each being supported.

```
<table>
  <caption>A summary of the planets of our solar system</caption>
  <thead></thead>
  <tbody></tbody>
  <tfoot></tfoot>
</table>
```

Table Content

- Tables can be divided into table headers, bodies, and footers, but none of these really does anything if the tables do not contain table rows, cells, and content.
- Each table row, `<tr>` contains one or more cells.
- If a cell is a header cell, use `<th>` . Otherwise, use `<td>` .

```
<table>
  <caption>A summary of the planets of our solar system</caption>
  <thead>
    <tr>
      <th>Name</th>
      <th>Mass</th>
      <th>Diameter</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Mercury</td>
      <td>0.055</td>
      <td>4,879</td>
    </tr>
    <tr>
      <td>Venus</td>
      <td>0.815</td>
      <td>12,104</td>
    </tr>
  </tbody>
  <tfoot>
  </tfoot>
</table>
```

Merging Cells

- Similar to MS Excel, Google Sheets, and Numbers, it is possible to join multiple cells into a single cell.
- The `colspan` attribute is used to merge two or more adjacent cells within a single row.
- The `rowspan` attribute is used to merge cells across rows, being placed on the cell in the top row.

```
<table>
  <caption>Employee Skills Matrix</caption>
  <thead>
    <tr>
      <th>Employee</th>
      <th>Skill</th>
      <th>Level</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td rowspan="3">John Smith</td>
      <td>JavaScript</td>
      <td>Advanced</td>
    </tr>
    <tr>
      <td>Python</td>
      <td>Intermediate</td>
    </tr>
    <tr>
      <td>SQL</td>
      <td>Beginner</td>
    </tr>
    <tr>
      <td rowspan="2">Sarah Johnson</td>
      <td>HTML/CSS</td>
      <td>Advanced</td>
    </tr>
    <tr>
      <td>React</td>
      <td>Intermediate</td>
    </tr>
```

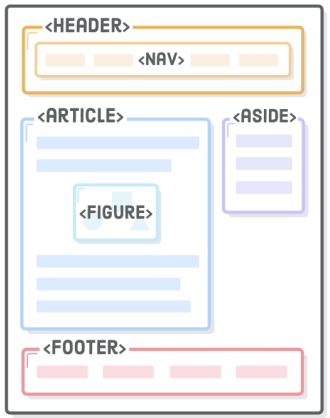
The Tag

- The `<div>` tag is a block-level container used to group and organize content in HTML.
- It has no semantic meaning by itself
- Commonly used with **CSS** for layout and styling
- Can wrap other elements like text, images, or entire sections
- Think of a `<div>` as a "box" to organize content.

```
<div class="card">
  <h2>Article Title</h2>
  <p>This is a paragraph inside a div container.</p>
</div>
```

Semantic HTML

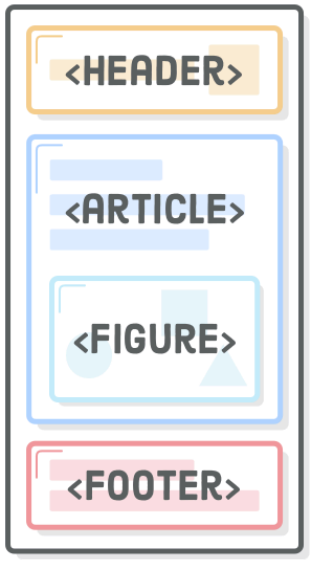
- With over 100 HTML elements and the ability to create custom elements, there are infinite ways to mark up your content. But, some ways—notably *semantically* —are better than others.
- Semantic HTML uses markup to convey the meaning of web content, not just its appearance, by employing elements like `<header>` , `<nav>` , `<main>` , `<article>` , `<section>` , `<aside>` , and `<footer>` .
- This practice enhances accessibility, SEO, and code readability. It includes proper use of headings, lists, and tables.
- Semantic HTML aids screen readers, improves browser rendering, and offers clearer structure for developers, leading to more meaningful, accessible, and maintainable web documents for both humans and machines.



Using Semantic HTML

AMBIGUOUS STRUCTURE

(AKA "<DIV> SOUP")



IDENTIFIABLE SECTIONS

(AKA "SEMANTIC MARKUP")

Using Semantic HTML

```
<div>
  <span>Three words</span>
  <div>
    <a>one word</a>
    <a>one word</a>
    <a>one word</a>
    <a>one word</a>
  </div>
</div>
<div>
  <div>
    <div>five words</div>
  </div>
  <div>
    <div>three words</div>
    <div>forty-six words</div>
    <div>forty-four words</div>
  </div>
  <div>
    <div>seven words</div>
    <div>sixty-eight words</div>
    <div>forty-four words</div>
  </div>
</div>
<div>
   <span>five words</span>
</div>
```

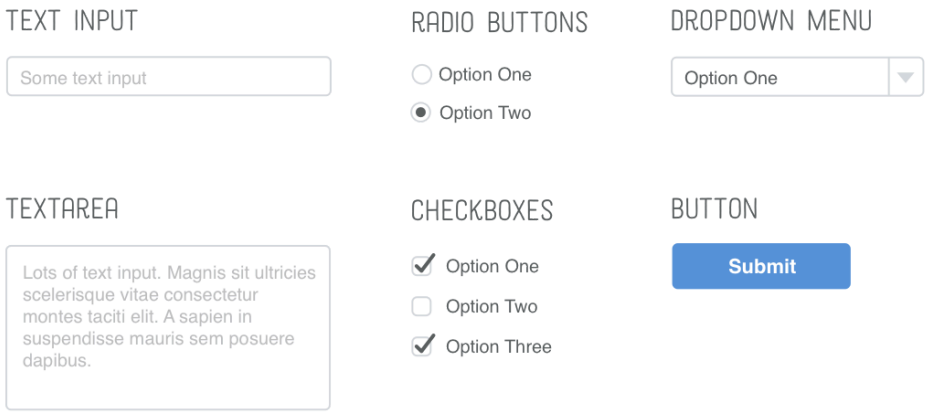
```
<header>
  <h1>Three words</h1>
  <nav>
    <a>one word</a>
    <a>one word</a>
    <a>one word</a>
    <a>one word</a>
  </nav>
</header>
<main>
  <header>
    <h2>five words</h2>
  </header>
  <section>
    <h2>three words</h2>
    <p>forty-six words</p>
    <p>forty-four words</p>
  </section>
  <section>
    <h3>seven words</h3>
    <p>sixty-eight words</p>
    <p>forty-four words</p>
  </section>
</main>
<footer>
  <p>five words</p>
</footer>
```

The second code example, with semantic elements, provides enough context for a non-coder to decipher the purpose and meaning without having ever encountered an HTML tag.

Common Semantic HTML Elements

Tag	Description
`<header>`	Represents introductory content, typically containing navigational links or a logo.
`<nav>`	Defines a set of navigation links.
`<main>`	Specifies the main content of a document, unique and central to the page.
`<section>`	Groups related content together, typically with a heading.
`<article>`	Represents a self-contained piece of content, such as a blog post or news article.
`<aside>`	Contains content that is tangentially related to the main content, like sidebars or ads.
`<footer>`	Represents the footer for a document or section, often containing copyright or links.
`<figure>`	Encapsulates a visual representation, such as an image or diagram, with an optional caption.
`<figcaption>`	Provides a caption for a `<figure>` element.
`<details>`	Creates a collapsible content section that users can toggle open or closed.
`<summary>`	Defines a visible heading for a `<details>` element.
`<time>`	Represents a specific time or date.
`<mark>`	Highlights text for reference purposes.
`<code>`	Represents a fragment of computer code.
`<address>`	Contains contact information, typically for the author of the document.

Forms



- HTML form elements let you **collect input** from your website's visitors.
- Mailing lists, contact forms, and blog post comments are common examples for small websites, but in organizations that rely on their website for revenue, forms are sacred and revered.

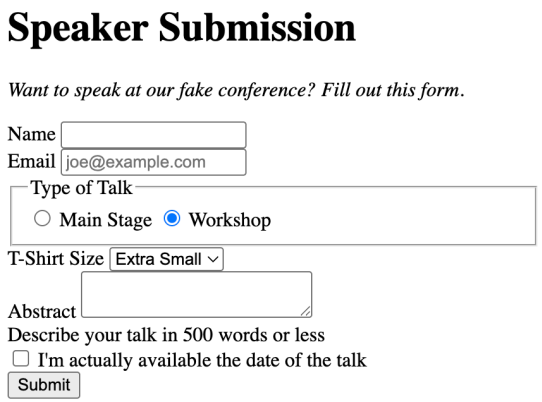
Forms are the money pages

- They're how e-commerce sites sell their products, how SaaS companies collect payment for their service, and how non-profit groups raise money online.
- Many companies measure the success of their website by the effectiveness of its forms because they answer questions like “how many leads did our website send to our sales team?” and “how many people signed up for our product last week?”
- This often means that forms are subjected to endless A/B tests and optimizations.

- There are two aspects of a functional HTML form: the frontend user interface and the backend server.
- The former is the **appearance** of the form (as defined by HTML and CSS),
- The latter is the code that **processes** it (storing data in a database, sending an email, etc).
- We'll be focusing entirely on the frontend in this module.

Setup

- This is a speaker submission form for a fake conference.
- It hosts a pretty good selection of HTML forms elements:
  - various types of text fields
  - a group of radio buttons
  - a dropdown menu
  - a checkbox
  - a submit button



Setup

```
<!DOCTYPE html>
<html lang='en'>
  <head>
    <meta charset='UTF-8'/>
    <title>Speaker Submission</title>
  </head>
  <body>
    <header class='speaker-form-header'>
      <h1>Speaker Submission</h1>
      <p>
        <em>
          Want to speak at our fake conference? Fill out
          this form.
        </em>
      </p>
    </header>
  </body>
</html>
```

HTML Forms

Every HTML form begins with the aptly named `<form>` element. It accepts a number of attributes , but the most important ones are `action` and `method` .

Go ahead and add an empty form to our HTML document, right under the `<header>` :

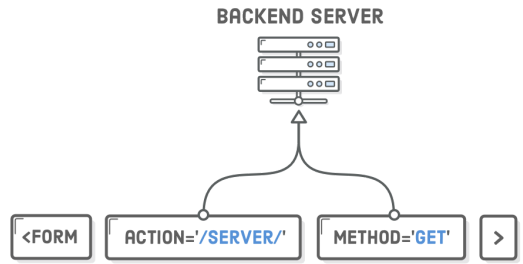
```
<form action='' method='get' class='speaker-form'>

</form>
```

The HTML form element identifies a document landmark containing interactive controls for submitting information. Nested in a `<form>` you'll find all the interactive (and non-interactive) form controls that make up that form.

Action and Method

- The `action` attribute defines the URL that processes the form. It's where the input collected by the form is sent when the user clicks the **Submit** button. This is typically a special URL defined by your web server that knows how to process the data.
- The `method` attribute can be either post or get , both of which define how the form is submitted to the backend server. This is largely dependent on how your web server wants to handle the form, but the general rule of thumb is to use `post` when you're *changing* data on the server, reserving `get` for when you're only *getting* data.
- By leaving the `action` attribute blank, we're telling the form to submit to the same URL. Combined with the `get` method, this will let us inspect the contents of the form.



Text Input Fields

```
<div>
  <label for='full-name'>Name</label>
  <input id='full-name' name='full-name' type='text'/>
</div>
```

First, we have a container `<div>` to help with styling. This is pretty common for separating input elements. Second, we have a `<label>` , which you can think of as another semantic HTML element , like `<article>` or `<figcaption>` , but for form labels. A label's `for` attribute must match the `id` attribute of its associated `<input/>` element.

Text Input Fields

Third, the `<input/>` element creates a text field. It's a little different from other elements we've encountered because it can dramatically change appearance depending on its `type` attribute, but it always creates some kind of interactive user input. The `id` attribute here is *only* for connecting it to a `<label>` element.

Conceptually, an `<input/>` element represents a "variable" that gets sent to the backend server. The `name` attribute defines the name of this variable, and the value is whatever the user entered into the text field. Note that you can pre-populate this value by adding a `value` attribute to an `<input/>` element.

Email Input Fields

The `<input/>` element's `type` attribute also lets you do basic input validation. For example, let's try adding another input element that *only* accepts email addresses instead of arbitrary text values:

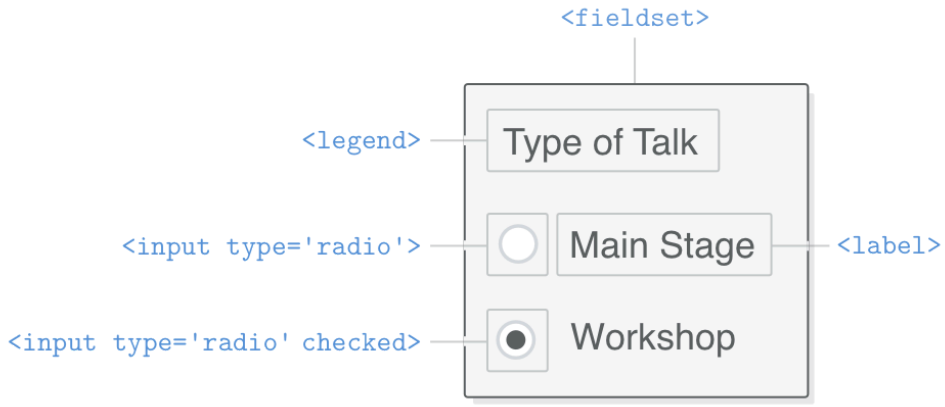
```
<div>
  <label for='email'>Email</label>
  <input id='email'
         name='email'
         type='email'
         placeholder='joe@example.com'/>
</div>
```

This works exactly like the `type='text'` input, except it automatically checks that user entered an email address. In Firefox, you can try typing something that's not an email address, then clicking outside of the field to make it lose focus and validate its input. It should turn red to show the user that it's an incorrect value. Chrome and Safari don't attempt to validate until user tries to submit the form, so we'll see this in action later in this chapter.

Also notice the new `placeholder` attribute that lets you display some default text when the `<input/>` element is empty. This is a nice little UX technique to prompt the user to input their own value.

Radio Buttons

Changing the `type` property of the `<input/>` element to `radio` transforms it into a radio button. Radio buttons are a little more complex to work with than text fields because they always operate in groups, allowing the user to choose one out of many predefined options.



Radio Buttons

This means that we not only need a label for each `<input/>` element, but also a way to group radio buttons and label the entire group. This is what the `<fieldset>` and `<legend>` elements are for. Every radio button group you create should:

- Be wrapped in a `<fieldset>` , which is labeled with a `<legend>` .
- Associate a `<label>` element with each radio button.
- Use the same `name` attribute for each radio button in the group.
- Use different `value` attributes for each radio button.

```
<fieldset>
  <legend>Type of Talk</legend>
  <input id='talk-type-1'
         name='talk-type'
         type='radio'
         value='main-stage' />
  <label for='talk-type-1'>Main Stage</label>
  <input id='talk-type-2'
         name='talk-type'
         type='radio'
         value='workshop'
         checked />
  <label for='talk-type-2'>Workshop</label>
</fieldset>
```

The user can't enter custom values into a radio button, which is why each one of them needs an explicit `value` attribute. This is the value that will get sent to the server when the user submits the form. It's also very important that each radio button has the same `name` attribute, otherwise the form wouldn't know they were part of the same group.

Select Elements (Dropdown Menus)

Dropdown menus offer an alternative to radio buttons, as they let the user select one out of many options. The `<select>` element represents the dropdown menu, and it contains a bunch of `<option>` elements that represent each item.

```
<div>
  <label for='t-shirt'>T-Shirt Size</label>
  <select id='t-shirt' name='t-shirt'>
    <option value='xs'>Extra Small</option>
    <option value='s'>Small</option>
    <option value='m'>Medium</option>
    <option value='l'>Large</option>
  </select>
</div>
```

Just like our radio button `<input/>` elements, we have `name` and `value` attributes that get passed to the backend server. But, instead of being defined on a single element, they're spread across the `<select>` and `<option>` elements.

Textareas

The `<textarea>` element creates a multi-line text field designed to collect large amounts of text from the user. They're suitable for things like biographies, essays, and comments. Go ahead and add a `<textarea>` to our form, along with a little piece of instructional text:

```
<div>
  <label for='abstract'>Abstract</label>
  <textarea id='abstract' name='abstract'></textarea>
  <div>Describe your talk in 500 words or less</div>
</div>
```

Note that this isn't self-closing like the `<input/>` element, so you always need a closing `</textarea>` tag. If you want to add any default text, it needs to go *inside* the tags opposed to a `value` attribute.

Checkboxes

Checkboxes are sort of like radio buttons, but instead of selecting only one option, they let the user pick as many as they want. This simplifies things, since the browser doesn't need to know which checkboxes are part of the same group. In other words, we don't need a `<fieldset>` wrapper or shared `name` attributes.

```
<div>
  <label for="available">
    <input id="available" name="available" type="checkbox" value="is-available" />
    <span>I'm actually available the date of the talk</span>
  </label>
</div>
```

The way we used `<label>` here was a little different than previous sections. Instead of being a separate element, the `<label>` wraps its corresponding `<input/>` element. This is perfectly legal, and it'll make it easier to match our desired layout. It's still a best practice to use the `for` attribute.

Submitting Forms

- Forms are submitted when the user activates a submit button nested within the form.
- When using `<input>` for buttons, the 'value' is the button's label, and is displayed in the button.
- When using `<button>` , the label is the text between the opening and closing `<button>` tags.
- A submit button can be written either of two ways:

```
<input type="submit" value="Submit Form">

<button type="submit">Submit Form</button>
```

Submitting Forms

```
<div>
  <button>Submit</button>
</div>
```

Clicking the button tells the browser to validate all of the `<input/>` elements in the form and submit it to the `action` URL if there aren't any validation problems. So, you should now be able to type in something that's not an email address into our email field, click the `<button>` , and see an error message.

Submitting Forms

Clicking the button tells the browser to validate all of the `<input/>` elements in the form and submit it to the `action` URL if there aren't any validation problems. So, you should now be able to type in something that's not an email address into our email field, click the `<button>` , and see an error message.

This also gives us a chance to see how the user's input gets sent to the server. First, enter some values into all the `<input/>` fields, making sure the email address validates correctly. Then, click the button and inspect the resulting URL in your browser. You should see something like this:

```
http://localhost:57954/?full-name=Carlos&email=cjsanchez%40uloyola.es&talk-type=workshop&t-shirt=m&abstract=Testing+this+out&available=is-available
```

Everything after the `?` represents the variables in our form. Each `<input/>` 's `name` attribute is followed by an equal sign, then its value, and each variable is separated by an `&` character. If we had a backend server, it'd be pretty easy for it to pull out all this information, query a database (or whatever), and let us know whether the form submission was successful or not.

Forms and Validations

- Before sending data from a form to a server, it's important to check if all required fields are filled in correctly.
- This is called *client-side form validation* .
- It helps make sure the data sent matches what the form expects, improving data quality and user experience.
- For example, you enter a username and submit a form—only to find out that usernames must have at least eight characters.

Help users enter the right data in forms

- Browsers have built-in features for validation to check that users have entered data in the correct format.
- You can activate these features by using the correct elements and attributes. On top of that, you can enhance form validation with CSS and JavaScript.
- You can use HTML to specify the correct format and constraints for data entered in your forms. You also need to specify which fields are mandatory.

Example

Try to submit this form without entering any data:

```
<main>
  <div class="wrapper">
    <form>
      <div>
        <label for="name">Name (required)</label>
        <input required type="text" id="name" name="name">
      </div>
      <button>Save</button>
    </form>
  </div>
</main>
```

```
button {
  margin-top: 1rem;
}
```

Do you see an error message attached to the `<input>` telling you that the field is required? This happens because of the `required` attribute.

Form elements attributes

One of the most significant features of form controls is the ability to validate most user data without relying on JavaScript. This is done by using validation attributes on form elements:

- `required` : Specifies whether a form field needs to be filled in before the form can be submitted.
- `minlength` and `maxlength` : Specifies the minimum and maximum length of textual data (strings).
- `min` , `max` , and `step` : Specifies the minimum and maximum values of numerical input types, and the increment, or step, for values, starting from the minimum.
- `type` : Specifies whether the data needs to be a number, an email address, or some other specific preset type.
- `pattern` : Specifies a regular expression that defines a pattern the entered data needs to follow.

Other HTML Input Types

MDN Web Docs - Input Types

SEO

- SEO (Search Engine Optimization) helps websites show up higher in search results (the process of making your site better for search engines).
- This means using good keywords, creating quality content, making sure your site is fast and mobile-friendly, and getting links from other trusted sites.
- Good navigation and design also help. Technical things like sitemaps and secure HTTPS are important too.
- SEO aims to get more visitors from search engines by improving your site both on the page and technically.
- Resources:
  - Google Search Central
  - SEO for 2025: The Complete Guide

References

- Learn HTML
- Web.dev
- FreeCodeCamp - Responsive Web Design
- Interneting is Hard
- MDN Web Docs
- MDN Web Docs - Form validation

Module 01 Exercise

Build a Personal HTML Page

- Set up a **Vite** project (no framework) to serve a single `index.html` page introducing yourself.
- Your page must include at least the following HTML tags:
  - `<header>` , `<main>` , and `<footer>`
  - At least one `<img>` , `<ul>` / `<ol>` , and `<a>`
  - Use semantic tags and inline formatting tags like `<strong>` and `<em>`
- The `<h1>` element inside `<header>` must contain your full name
- Include HTML comments to document each section
- Ensure your HTML is valid and well-structured
- All configuration files so the server can be launched with: `npm install` and `npm run dev`

How to Submit

- Push your work to your GitHub repository under `exercises/01/`
- **To run:** reviewers must be able to: `cd exercises/01` , `npm install` , `npm run dev` and then access the page at `http://localhost:5173`
- **Deadline:** To be announced