THE HTML HANDBOOK

Flavio Copes

Table of Contents

Introduction

Welcome!

I wrote this book to help you quickly learn HTML and get familiar with the advanced HTML topics.

HTML, a shorthand for Hyper Text Markup Language, is one of the most fundamental building blocks of the Web.

HTML was officially born in 1993 and since then it evolved into its current state, moving from simple text documents to powering rich Web Applications.

This handbook is aimed at a vast audience.

First, the beginner. I explain HTML from zero in a succinct but comprehensive way, so you can use this book to learn HTML from the basics.

Then, the professional. HTML is often considered like a secondary thing to learn. It might be given for granted.

Yet lots of things are obscure to many people. Me included. I wrote this handbook to help my understanding of the topic, because when I need to

explain something, I better makesure I first know the thing inside out.

Even if you don't write HTML in your day to day work, knowing how HTML works can help save you some headaches when you need to understand it from time to time, for example while tweaking a web page.

Flavio

You can reach me via email at flavio@flaviocopes.com, on Twitter @flaviocopes.

My website is flaviocopes.com.

Preface

Preface

HTML is the foundation of the marvel called the Web.

There is an incredible power underneath this rather simple and limited set of rules, which lets us -- developers, makers, designers, writers, and tinkerers -- craft documents, apps, and experiences for people all around the globe.

My first HTML book came out in 1997 and was called "HTML Unleashed". A big, lots-of-pages, long tome.

20+ years have passed, and HTML is still the foundation of the Web, with minimal changes from back then.

Sure, we got more semantic tags, presentational HTML is no longer a thing, and CSS has taken care of the design of things.

HTML's success is based on one thing: **simplicity**.

It resisted being hijacked into an XML dialect via XHTML, when eventually people realized that thing was way, way too complex.

It did so because of another feature it provides us: **forgiveness**. There are *some* rules, right, but after you learn those, you have a lot of freedom.

Browsers learned to be resilient and to always try to do their best when parsing and presenting HTML to the users.

And the whole Web platform did one thing right: it never broke backward compatibility. Pretty incredibly, we can go back to HTML documents written in 1991, and they look pretty much as they looked back then.

We even know what the first web page was. It's this: http://info.cern.ch/hypertext/WWW/TheProject.html

And you can see the source of the page, thanks to another big feature of the Web and HTML: **we can inspect the HTML of any web page**.

Don't take this for granted. I don't know any other platform that gives us this ability.

The exceptional Developer Tools built into any browser let us inspect and take inspiration from HTML written by anyone in the world.

If you are new to HTML this book aims to help you get started. If you are a seasoned Web Developer this book will improve your knowledge.

I learned so muchwhile writing it, even though I've been working with the Web for 20+ years, and I'm sure you'll find something new, too.

Or you'll re-learn something old you forgot.

In any case, the goal of the book is to be useful to you, and I hope it succeeds.

HTML Basics

HTML Basics

HTML is a standard defined by the **WHATWG**, an acronym for Web Hypertext Application Technology Working Group, an organization formed by people working on the most popular web browser. This means it's basically controlled by Google, Mozilla, Apple and Microsoft.

In the past the **W3C** (World Wide Web Consortium) was the organization in charge of creating the HTML standard.

The control informally moved from W3C to WHATWG when it became clear that the W3C push towards XHTML was not a good idea.

If you've never heard of XHTML, here's a short story. In the early 2000s, we all believed the future of the Web was XML (seriously). So HTML moved from being an SGML-based authoring language to an XML markup language.

It was a big change. We had to know, and respect, more rules. Stricter rules.

Eventually browser vendors realized this was not the right path for the Web, and they pushed back, creating what is now known as HTML5.

W3C did not really agree on giving up control of HTML, and for years we had 2 competing standards, each one aiming to be the official one. Eventually on 28 May 2019 it was made official by W3C that the "true" HTML version was the one published by WHATWG.

I mentioned HTML5. Let me explain this little story. I know, it's kind of confusing up to now, as with many things in life when many actors are involved, yet it's also fascinating.

We had **HTML version 1** in 1993. Here's the original RFC.

HTML 2 followed in 1995.

We got **HTML 3** in January 1997, and **HTML 4** in December 1997.

Busy times!

20+ years went by, we had this entire XHTML thing, and eventually we got to this HTML5 "thing", which is not really *just HTML* any more.

HTML5 is a term that now defines a whole set of technologies, which includes HTML but adds a lot of APIs and standards like WebGL, SVG and more.

The key thing to understand here is this: there is no such thing (any more) as an HTML version now. It's a living standard. Like CSS, which is called "3", but in reality is a bunch of independent modules developed separately. Like JavaScript, where we have one new edition each year, but nowadays, the only thing that matters is which individual features are implemented by the engine.

Yes we call it HTML5, but HTML4 is from 1997. That's a long time for anything, let alone for the web.

This is where the standard now "lives": https://html.spec.whatwg.org/multipage.

HTML is the markup language we use to structure content that we consume on the Web.

HTML is served to the browser in different ways.

- It can be generated by a server-side application that builds it depending on the request or the session data, for example a Rails or Laravel or Django application.
- It can be generated by a JavaScript client-side application that generates HTML on the fly.
- In the simplest case, it can be stored in a file and served to the browser by a Web server.

Let's dive into this last case. Although in practice it's probably the least popular way to generate HTML, it's still essential to know the basic building blocks.

By convention, an HTML file is saved with a `.html` or `.htm` extension.

Inside this file, we organize the content using **tags**.

Tags wrap the content, and each tag gives a special meaning to the text it wraps.

Let's make a few examples.

This HTML snippet creates a paragraph using the `p` tag:

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

This HTML snippet creates a list of items using the `ul` tag, which means *unordered list*, and the `li` tags, which mean *list item*:

```
<ul>
    <li>First item</li>
    <li>Second item</li>
    <li>Third item</li>
</ul>
```

When an HTML page is served by the browser, the tags are interpreted, and the browser renders the elements according to the rules that define their visual appearance.

Some of those rules are built-in, such as how a list renders or how a link is underlined in blue.

Some other rules are set by you with CSS.

HTML is not presentational. It's not concerned with how things *look*.
Instead, it's concerned with what things *mean*.

It's up to the browser to determine how things look, with the directives defined by who builds the page, with the CSS language.

Now, those two examples I made are HTML snippets taken outside of a page context.

HTML page structure

Let's make an example of a proper HTML page.

Things start with the Document Type Declaration (aka *doctype*), a way to tell the browser this is an HTML page, and which version of HTML we are using.

Modern HTML uses this doctype:

```
<!DOCTYPE html>
```

Then we have the `html` element, which has an opening and closing tag:

```
<!DOCTYPE html>
<html> ...
</html>
```

Most tags come in pairs with an opening tag and a closing tag. The closing tag is written the same as the opening tag, but with a `/`:

```
<sometag>some content</sometag>
```

There are a few self-closing tags, which means they don't need a separate closing tag as they don't contain anything *in them*.

The `html` starting tag is used at the beginning of the document, right after the document type declaration.

The `html` ending tag is the last thing present in an HTML document.

Inside the `html` element we have 2 elements: `head` and `body`:

```
<!DOCTYPE html>
<html>

    <head>

    ...
                </head>

    <body>

    ...
                </body>
</html>
```

Inside `head` we will have tags that are essential to creating a web page, like the title, the metadata, and internal or external CSS and JavaScript. Mostly things that do not directly appear on the page, but only help the browser (or bots like the Google search bot) display it properly.

Inside `body` we will have the content of the page. The **visible stuff**.

Tags vs elements

I mentioned tags and elements. What's the difference?

Elements have a starting tag and a closing tag. In this example, we use the `p` starting and closing tags to create a `p` element:

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

So, an element constitutes the whole

- *package*: starting tag text
- content (and possibly other
- elements) closing tag

If an element has doesn't have a closing tag, it is only written with the startingtag, and it cannot contain any text content.

That said, I mightuse the tag or element term in the book meaning the same thing,except if I explicitly mention starting tag or ending tag.

Attributes

The starting tag of an element can have special snippets of information we can attach, called **attributes**.

Attributes have the `key="value"` syntax:

```
<p class="a-class">A paragraph of text</p>
```

> You can also use single quotes, but using double quotes in HTML is a nice convention.

We can have many of them:

```
<p class="a-class" id="an-id">A paragraph of text</p>
```

and some attributes are boolean, meaning you only need the key:

```
<script defer src="file.js"></script>
```

The `class` and `id` attributes are two of the most common you will find used.

They have a special meaning, and they are useful both in CSS and JavaScript.

The difference between the two is that an `id` is unique in the context of a web page; it cannot be duplicated.

Classes, on the other hand, can appear multiple times on multiple elements.

Plus, an `id` is just one value. `class` can hold multiple values, separated by a space:

```
<p class="a-class another-class">A paragraph of text</p>
```

It's common to use the dash `-` to separate words in a class value, but it's just a convention.

Those are just two of the possible attributes you can have. Some attributes are only used for one tag. They are highly specialized.

Other attributes can be used in a more general way. You just saw `id` and `class`, but we have other ones too, like `style` which can be used to insert inline CSS rules on an element.

Case insensitive

HTML is case insensitive. Tags can be written in all caps, or lowercase. In the early days, caps were the norm. Today lowercase is the norm. It is a convention.

You usually write like this:

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

not like this:

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

White space

Pretty important. In HTML, even if you add multiple white spaces into a line, it's collapsed by the browser's CSS engine.

For example the rendering of this paragraph

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

is the sameas this:

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

and the sameas this:

```
<p>A paragraph

of

                                        text
                                    </p>
```

> Using the `white-space` CSS property you can change how things behave. You can find more information on how CSS processes white space in the CSS Spec

I'd say use the syntax that makes things visually more organized and easier to read, but you can use any syntax you like.

I typically favor

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

or

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

Nested tags should be indented with 2 or 4 characters, depending on your preference:

```
<body>
    <p>
        A paragraph of text
    </p>
    <ul>
        <li>A list item</li>
    </ul>
</body>
```

> Note: this means that if you want to add additional space, it can make you pretty mad. I suggest you use CSS to make more space when needed.
>
> Note: in special cases, you can use the ` ` HTML entity (an acronym that means *non-breaking space*) - more on HTML entities later on. I think this should not be abused. CSS is always preferred to alter the visual presentation.

The document heading

The document heading

The `head` tag contains special tags that define the document properties.

It's always written before the `body` tag, right after the opening `html` tag:

```
<!DOCTYPE html>
<html>

    <head>

        ...

    </head>

        ...
</html>
```

We never use attributes on this tag. And we don't write content in it.

It's just a container for other tags. Inside it we can have a wide variety of tags, depending on what you need to do:

- `title`
- `script`
- `noscript`
- `link`
- `style`
- `base`
- `meta`

The `title` tag

The `title` tag determines the page title. The title is displayed in the browser, and it's especially important as it's one of the key factors for Search Engine Optimization (SEO).

The `script` tag

This tag is used to add JavaScript into the page.

You can include it inline, using an opening tag, the JavaScript code and then the closing tag:

```
<script>
..some JS
</script>
```

Or you can load an

external JavaScript file by using the `src` attribute:

```
<script src="file.js"></script>
```

> The `type` attribute by default is set to `text/javascript`, so it's completely optional.

There is something pretty important to know about this tag.

Sometimes this tag is used at the bottom of the page, just before the closing `</body>` tag. Why? For performance reasons.

Loading scripts by default blocks the rendering of the page until the script is parsed and loaded.

By putting it at the bottom of the page, the script is loaded and executed after the whole page is already parsed and loaded, giving a better experience to the user over keeping it in the `head` tag.

My opinion is that this is now bad practice. Let `script` live in the `head` tag.

In modern JavaScript we have an alternative this is more performant than keeping the script at the bottom of the page -- the `defer` attribute. This is an example that loads a `file.js` file, relative to the current URL:

```
<script defer src="file.js"></script>
```

This is the scenario that triggers the faster path to a fast-loading page, and fast-loading JavaScript.

> Note: the `async` attribute is similar, but in my opinion a worse option than `defer`. I describe why, in more detail, on page https://flaviocopes.com/javascript-async-defer/

The `noscript` tag

This tag is used to detect when scripts are disabled in the browser.

> Note: users can choose to disable JavaScript scripts in the browser settings. Or the browser might not support them by default.

It is used differently depending on whether it's put in the document head or in the document body.

We're talking about the document head now, so let's first introduce this usage.

In this case, the `noscript` tag can only contain other tags:

- `link`
- tags `style`
- tags `meta`
- tags

to alter the resources served by the page, or the `meta` information, if scripts are disabled.

In this example I set an element with the `no-script-alert` class to display if scripts are disabled, as it was `display: none` by default:

```
<!DOCTYPE html>
<html>

    <head>

        ...

                                    <noscript>

        <style>

                                .no-script-
alert {

                    display: block;
```

```
            }
        </style>
                                                    </noscript>
                                                    ...

        </head>
            ...
</html>
```

> Let's solve the other case: if we put in the body, it can contain content, like paragraphs and other tags, which are rendered in the UI.

The `link` tag

The `link` tag is used to set relationships between a document and other resources.

It's mainly used to link an external CSS file to be loaded.

This element has no closing tag.

Usage:

```
<!DOCTYPE html>
<html>
    <head>
    ...
    <link href="file.css" rel="stylesheet">
    ...
    </head>
    ...
</html>
```

The `media` attribute allows the loading of different stylesheets depending on the device capabilities:

```
<link href="file.css" media="screen" rel="stylesheet">
<link href="print.css" media="print" rel="stylesheet">
```

We can also link to resources other than stylesheets.

For example we can associate an RSS feed using

```
<link rel="alternate" type="application/rss+xml" href="/index.xml"
>
```

Or we can associate a favicon using:

```
<link rel="apple-touch-icon" sizes="180x180" href="/assets/appletouch-icon.png">

<link rel="icon" type="image/png" sizes="32x32" href="/assets/fav icon-32x32.png">

<link rel="icon" type="image/png" sizes="16x16" href="/assets/fav icon-16x16.png">
```

This tag *was* also used for multi-page content, to indicate the previous and next page using `rel="prev"` and `rel="next"`. Mostly for Google. Asof 2019, Google announcedit does not use this tag any more because it can find the correct page structure without it.

The `style` tag

This tag can be used to add styles into the document, rather than loading an external stylesheet.

Usage:

```
<style>
.some-css {}
</style>
```

As with the `link` tag, you can use the `media` attribute to use that CSS only on the specified medium:

```
<style media="print">
.some-css {}
</style>
```

The `base` tag

This tag is used to set a base URL for all relative URLs contained in the page.

```
<!DOCTYPE html>
<html>
  <head>
    ...
    <base href="https://flaviocopes.com/">
    ...
  </head>
  ...
</html>
```

The `meta` tag

Meta tags perform a variety of tasks and they are very, very important.

Especially for SEO.

`meta` elements only have the starting tag.

The most basic one is the `description` meta tag:

```
<meta name="description" content="A nice page">
```

This *might* be used by Google to generate the page description in its result pages, if it finds it better describes the page than the on-page content (don't ask me how).

The `charset` meta tag is used to set the page character encoding. `utf-8` in most cases:

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

The `robots` meta tag instructs the Search Engine bots whether to index a page or not:

```
<meta name="robots" content="noindex">
```

Or if they should follow links or not:

```
<meta name="robots" content="nofollow">
```

> You can set nofollow on individual links, too. This is how you can set `nofollow` globally.

You can combine them:

```
<meta name="robots" content="noindex, nofollow">
```

The default behavior is `index, follow`.

You can use other properties, including `nosnippet`, `noarchive`, `noimageindex` and more.

You can also just tell Google instead of targeting *all* search engines:

```
<meta name="googlebot" content="noindex, nofollow">
```

And other search engines mighthave their own meta tag, too.

Speaking of which, we can tell Google to disable somefeatures. This prevents the translate functionality in the search engine results:

```
<meta name="google" content="notranslate">
```

The `viewport` meta tag is used to tell the browser to set the page width based on the device width.

```
<metaname="viewport" content="width=device-width, initial-scale=1">
```

See more on this tag.

Another rather popular meta tag is the `http-equiv="refresh"` one.

This line tells the browser to wait 3 seconds, then redirect to that other page:

```
<meta http-equiv="refresh" content="3;url=http://flaviocopes.com/ another-page">
```

Using 0 instead of 3 will redirect as soon as possible.

This is not a full reference; Other less-used meta tags exist.

After this document heading introduction, we can start diving into the document body.

The document body

The document body

After the closing head tag, we can only have one thing in an HTML document: the `body` element.

```
<!DOCTYPE html>
<html>

    <head>

    ...
    </head>

    <body>

    ...
    </body>
</html>
```

Just like the `head` and `html` tags, we can only have one `body` tag in one page.

Inside the `body` tag we have all the tags that define the content of the page.

Technically, the start and ending tags are optional. But I consider it a good practice to add them. Just for clarity.

In the next chapters we'll define the variety of tags you can use inside the page body.

But before, we must introduce a difference between block elements and inline elements.

Block elements vs inline elements

Visual elements, the ones defined in the page body, can be generally classified in 2 categories:

- block elements (`p`, `div`, heading elements, lists and list items, ...)
- inline elements (`a`, `span`, `img`, ...)

What is the difference?

Block elements, when positioned in the page, do not allow other elements next to them. To the left, or to the right.

Inline elements instead can sit next to other inline elements.

The difference also lies in the visual properties we can edit using CSS. We can alter the width/height, margin, padding and border of block elements. We can't do that for inline elements.

> Note that using CSS we can change the default for each element, setting a `p` tag to be inline, for example, or a `span` to be a block element.

Another difference is that inline elements can be contained in block elements. The reverse is not true.

Some block elements can contain other block elements, but it depends. The `p` tag for example does not allow such option.

Tags that interact with text

Tags that interact with text

The `p` tag

This tag defines a paragraph of text.

```
<p>Some text</p>
```

It's a block element.

Inside it, we can add

any inline element we like, like `span` or `a`.

We cannot add block elements.

We cannot nest a `p` element into another one.

By default browsers style a paragraph with a margin on top and at the bottom. `16px` in Chrome, but the exact value might vary between browsers.

This causes two consecutive paragraphs to be spaced, replicating what we think of a "paragraph" in printed text.

The `span` tag

This is an inline tag that can be used to create a section in a paragraph that can be targeted using CSS:

```
<p>A part of the text <span>and here another part</span></p>
```

The `br` tag

This tag represents a line break. It's an inline element, and does not need a closing tag.

We use it to create a new line inside a `p` tag, without creating a new paragraph.

And compared to creating a new paragraph, it does not add additional spacing.

```
<p>Some text<br>A new line</p>
```

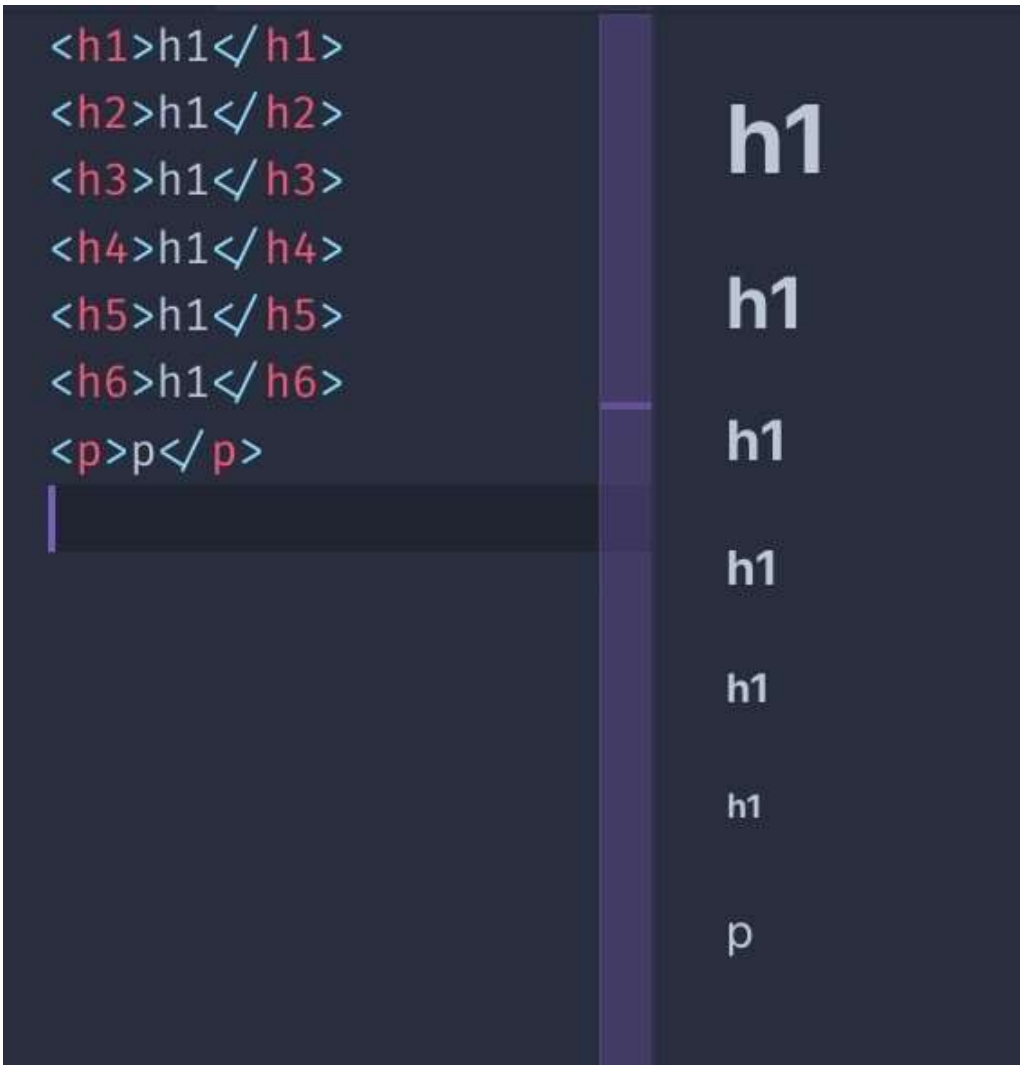
The heading tags

HTML provides us 6 heading tags. From most important to least important, we have `h1`, `h2`, `h3`, `h4`, `h5`, `h6`.

Typically a page will have one `h1` element, which is the page title. Then you might have one or more `h2` elements depending on the page content.

Headings, especially the heading organization, are also essential for SEO, and search engines use them in various ways.

The browser by default will render the `h1` tag bigger, and will make the elements size smaller as the number near `h` increases:



All headings are block elements. They cannot contain other elements, just text.

The `strong` tag

This tag is used to mark the text inside it as *strong*. This is pretty important, it's not a visual hint, but a semantic hint. Depending on the medium used, its interpretation will vary.

Browsers by default make the text in this tag **bold**.

The `em` tag

This tag is used to mark the text inside it as *emphasized*. Like with `strong`, it's not a visual hint but a semantic hint.

Browsers by default make the text in this **italic**.

Quotes

The `blockquote` HTML tag is useful to insert citations in the text.

Browsers by default apply a margin to the `blockquote` element. Chrome applies a 40px left and right margin, and a 10px top and bottom margin.

The `q` HTML tag is used for inline quotes.

Horizontal line

Not really based on text, but the `hr` tag is often used inside a page. It means `horizontal rule`, and it adds a horizontal line in the page.

Useful to separate sections in the page.

Code blocks

The `code` tag is especially useful to show code, because browsers give it a monospaced font.

That's typically the only thing that browsers do. This is the CSS applied by Chrome:

```
code{
    font-family: monospace; }
```

This tag is typically wrapped in a `pre` tag, because the `code` element ignores whitespace and line breaks. Like the `p` tag.

Chrome gives `pre` this default styling:

```
pre {
    display: block;
    font-family: monospace;
    white-space: pre;
    margin: 1em 0px; }
```

which prevents white space collapsing and makes it a block element.

Lists

We have 3 types of lists:

- unordered lists
- ordered lists
- definition lists

Unordered lists are created using the `ul` tag. Each item in the list is created with the `li` tag:

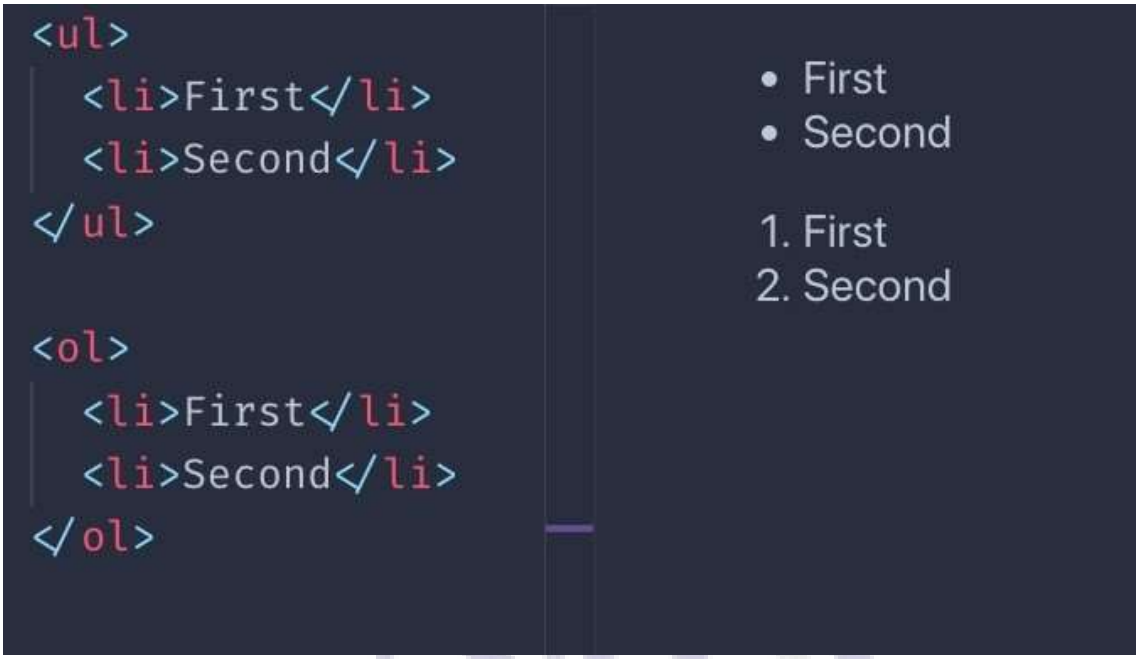
```
<ul>
    <li>First</li>
    <li>Second</li>
</ul>
```

Ordered lists are similar, just made with the `ol` tag:

```
<ol>
    <li>First</li>
    <li>Second</li>
```

```
</ol>
```

The difference between the two is that ordered lists have a number before each item:



Definition lists are a bit different. You have a term, and its definition:

```
<dl>
    <dt>Flavio</dt>
    <dd>The name</dd>
    <dt>Copes</dt>
    <dd>The surname</dd>
</dl>
```

This is how browsers typically render them:

I must say you rarely see them in the wild, for sure not much as `ul` and `ol`, but sometimes they might be useful.

Other text tags

There is a number of tags with presentational purposes:

- the
- `mark` tag
- the `ins`
- tag the
- `del`
- tag the
- `sup` tag

the `sub`

tag the

`small` tag

the `i`

tag

- the `b` tag

This is an example of the visual rendering of them which is applied by default by browsers:

You might wonder, how is `b` different than `strong`? And how `i` is different than `em`?

The difference lies in the semantic meaning. While `b` and `i` are a direct hint at the browser to make a piece of text bold or italic, `strong` and `em` give the text a special meaning, and it's up to the browser to give the styling. Which happens to be exactly the same as `b` and `i`, by default. Although you can change that using CSS.

There are a number of other, less used tags related to text. I just mentioned the ones that I see used the most.

Links

Links

Links are defined using the `a` tag. The link destination is set via its `href` attribute.

Example:

```
<a href="https://flaviocopes.com">click here</a>
```

Between the starting and closing tag we have the link text.

The above example is an absolute URL. Links also work with relative URLs:

```
<a href="/test">click here</a>
```

In this case, when clicking the link the user is moved to the `/test` URL on the current origin.

Be careful with the `/` character. If omitted, instead of starting from the origin, the browser will just add the `test` string to the current URL.

Example, I'm on the page `https://flaviocopes.com/axios/` and I have these links:

- `/test` once clicked brings me to `https://flaviocopes.com/axios/test`
- `https://flaviocopes.com/test` once clicked brings me to `test`

Links

Link tags can include other things inside them, not just text. For example, images:

```
<a href="https://flaviocopes.com"><img src="test.jpg"></a>
```

or any other elements, except other `<a>` tags.

If you want to open the link in a new tab, you can use the `target` attribute:

```
<a href="https://flaviocopes.com" target="_blank">open in new tab</a>
```

Container tags and page structure HTML

Container tags and page structure HTML

Container tags

HTML provides a set of container tags. Those tags can contain an unspecified set of other tags.

We have:

- article
- section
- div

and it can be confusing to understand the difference between them.

Let's see when to use each one of them.

article

The article tag identifies a *thing* that can be independent from other *things* in a page.

For example a list of blog posts in the homepage.

Or a list of links.

```
<div>
                    <article>

                <h2>A blog
        post</h2>
                                <a
        ...>Read        more</a>
                    </article>
                    <article>

            <h2>Another blog post</h2>

            <a      ...>Read        more</a>
                    </article>
</div>
```

We're not limited to lists: an article can be the main element in a page.

```
<article>
                <h2>A blog post</h2>                    <p>Here
    is      the     content...</p>
</article>
```

Inside an `article` tag we should have a title (`h1` - `h6`) and

section

Represents a section of a document. Each section has a heading tag (`h1` - `h6`), then the section *body*.

Example:

```
<section>                    <h2>A section of the page</h2>
                <p>...</p>
                <img ...>
</section>
```

It's useful to break a long article into different **sections**.

Shouldn't be used as a generic container element. `div` is made for this.

div

`div` is the generic container element:

```
<div>                        ...
</div>
```

You often add a `class` or `id` attribute to this element, to allow it to be styled using CSS.

We use `div` in any place where we need a container but the existing tags are not suited.

Tags related to page

nav

This tag is used to create the markup that defines the page navigation.

Into this we typically add an `ul` or `ol` list:

```
<nav>
        <ol>
                <li><a href="/">Home</a></li>
                <li><a href="/blog">Blog</a></li>
        </ol>
</nav>
```

aside

The `aside` tag is used to add a piece of content that is related to the main content.

A box where to add a quote, for example. Or a sidebar.

Example:

```
<div>   <p>some
    text..</p>
        <aside>

    <p>A quote..</p>
        </aside>
        <p>other
    text...</p>
</div>
```

Using `aside` is a signal that the things it contains are not part of the regular flow of the section it lives into.

header

The `header` tag represents a part of the page that is the introduction. It can for example contain one or more heading tag (`h1` - `h6`), the tagline for the article, an image.

```
<article>
        <header>
    <h1>Article title</h1>
    </header>
        ...
</div>
```

main

The `main` tag represents the main part of a page:

```
<body>
    ....
        <main>
            <p>....</p>
        </main>
</body>
```

footer

The `footer` tag is used to determine the footer of an article, or the footer of the page:

```
<article>          ....
                   <footer>
                            <p>Footer
          notes..</p>
                   </footer>
</div>
```

Forms

Forms

Forms are the way you can interact with a page, or an app, built with Web technologies.

You have a set of controls, and when you submit the form, either with a click to a "submit" button or programmatically, the browser will send the data to the server.

By default this data sending causes the page to reload after the data is sent, but using JavaScript you can alter this behavior (not going to explain how in this book).

A form is created using the `form` tag:

```
<form>
...
</form>
```

By default forms are submitted using the GET HTTP method. Which has its drawbacks, and usually you want to use POST.

You can set the form to use POST when submitted by using the `method` attribute:

```
<form method="POST">
  ...
</form>
```

The form is submitted, either using GET or POST, to the same URL where it resides.

So if the form is in the page `https://flaviocopes.com/contacts`, pressing the "submit" button will make a request to that same URL.

Which might result in nothing happening.

You need something server-side to handle the request, and typically you "listen" for those form submit events on a dedicated URL.

You can specify the URL via the `action` parameter:

```
<form action="/new-contact" method="POST">
  ...
</form>
```

This will cause the browser to submit the form data using POST to the `/new-contact` URL on the same origin.

If the origin (protocol + domain + port) is `https://flaviocopes.com`

(port 80 is the default), this means the form data will be sent to

`https://flaviocopes.com/new-contact`.

I talked about data. Which data?

Data is provided by users via the set of controls that are available on the Web platform:

- input boxes (single line text)
- text areas (multiline text)
- select boxes (choose one option from a drop-down menu)
- radio buttons (choose one option from a list always visible)
- checkboxes (choose zero, one or more option)
- file uploads and more!

Let's introduce each one of them in the following form fields overview.

The `input` tag

The `input` field is one of the most widely used form elements. It's also a very versatile element, and it can completely change behavior based on the `type` attribute.

The default behavior is to be a single-line text input control:

```
<input>
```

Equivalent to using:

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

As with all the other fields that follow, you need to give the field a name in order for its content to be sent to the server when the form is submitted:

```
<input type="text" name="username">
```

The `placeholder` attribute is used to have some text showing up, in light gray, when the field is empty. Useful to add a hint to the user for what to type in:

```
<input type="text" name="username" placeholder="Your username">
```

Email

Using `type="email"` will validate client-side (in the browser) an email for correctness (semantic correctness, not ensuring the email address is existing) before submitting.

```
<input type="email" name="email" placeholder="Your email">
```

Password

Using `type="password"` will make every key entered appear as an asterisk (*) or dot, useful for fields that host a

```
<input type="password" name="password" placeholder="Your password"
>
```

Numbers

You can have an input element accept only numbers:

```
<input type="number" name="age" placeholder="Your age">
```

You can specify a minimum and maximum value accepted:

```
<input type="number" name="age" placeholder="Your age" min="18"
    m ax="110">
```

The `step` attribute helps identify the steps between different values.

For example this accepts a value between 10 and 50, at steps of 5:

```
<input type="number" name="a-number" min="10" max="50" step="5">
```

Hidden field

Fields can be hidden from the user. They will still be sent to the server upon the form submit:

```
<input type="hidden" name="some-hidden-field" value="some-value">
```

This is commonly used to store values like a CSRF token, used for security and user identification, or even to detect robots sending spam, using special techniques.

It can also just be used to identify a form and its action.

Setting a default value

All those fields accept a predefined value. If the user does not change it, this will be the value sent to the server:

```
<input type="number" name="age" value="18">
```

If you set a placeholder, that value will appear if the user clears the input field value:

```
<input type="number" name="age" placeholder="Your age" value="18">
```

Form submit

The `type="submit"` field is a button that, once pressed by the user, submits the form:

```
<input type="submit">
```

The `value` attribute sets the text on the button, which if missing shows the "Submit" text:

```
<input type="submit" value="Click me">
```

Form validation

Browsers provide client-side validation functionality to forms.

You can set fields as required, ensuring they are filled, and enforce a specific format for the input of each field.

Let's see both options.

Set fields as required

The `required` attribute helps you with validation. If the field is not set, client-side validation fails and the browser does not submit the form:

```
<input type="text" name="username" required>
```

Enforce a specific format

I described the `type="email"` field above. It automatically validates the email address according to a format set in the specification.

In the `type="number"` field, I mentioned the `min` and `max` attribute to limit values entered to an interval.

You can do more.

You can enforce a specific format on any field.

The `pattern` attribute gives you the ability to set a regular expression to validate the value against.

I recommend reading my Regular Expressions Guide at flaviocopes.com/javascript-regular-expressions/.

pattern="https://.*"

```
<input type="text" name="username" pattern="[a-zA-Z]{8}">
```

Otherfields

File uploads

You can load files from your local computer and send them to the server using a `type="file"` input element:

```
<input type="file" name="secret-documents">
```

You can attach multiple files:

```
<input type="file" name="secret-documents" multiple>
```

You can specify one or more file types allowed using the `accept` attribute. This accepts images:

```
<input type="file" name="secret-documents" accept="image/*">
```

You can use a specific MIME type, like `application/json` or set a file extension like `.pdf`. Or set multiple file extensions, like this:

```
<input type="file" name="secret-documents" accept=".jpg, .jpeg, .png">
```

Buttons

The `type="button"` input fields can be used to add additional buttons to the form, that are not submit buttons:

```
<input type="button" value="Click me">
```

They are used to programmatically do something, using JavaScript.

There is a special field rendered as a button, whose special action is to clear the entire form and bring back the state of the fields to the initial one:

```
<input type="reset">
```

Radio buttons

Radio buttons are used to create a set of choices, of which one is pressed and all the others are disabled.

The name comes from old car radios that had this kind of interface.

You define a set of `type="radio"` inputs, all with the same `name` attribute, and different `value` attribute:

```
<input type="radio" name="color" value="yellow">
<input type="radio" name="color" value="red">
<input type="radio" name="color" value="blue">
```

Once the form is submitted, the `color` data property will have one single value.

There's always one element checked. The first item is the one checked by default.

You can set the value that's pre-selected using the `checked` attribute.

You can use it only once per radio inputs group.

Checkboxes

Similar to radio boxes, but they allow multiple values to be chosen, or none at all.

You define a set of `type="checkbox"` inputs, all with the same `name` attribute, and different `value` attribute:

```
<input type="checkbox" name="color" value="yellow">
<input type="checkbox" name="color" value="red">
<input type="checkbox" name="color" value="blue">
```

All those checkboxes will be unchecked by default. Use the `checked` attribute to enable them on page load.

Since this input field allows multiple values, upon form submit the value(s) will be sent to the server as an array.

Date andtime

We have a few input types to accept date values.

The `type="date"` input field allows the user to enter a date, and shows a date picker if needed:

```
<input type="date" name="birthday">
```

The `type="time"` input field allows the user to enter a time, and shows a time picker if needed:

```
<input type="time" name="time-to-pickup">
```

The `type="month"` input field allows the user to enter a month and a year:

```
<input type="month" name="choose-release-month">
```

The `type="week"` input field allows the user to enter a week and a year:

```
<input type="week" name="choose-week">
```

All those fields allow to limit the range and the step between each value. I recommend checking MDN for the little details on their usage.

The `type="datetime-local"` field lets you choose a date and a time.

```
<input type="datetime-local" name="date-and-time">
```

Here is a page to test them all:
https://codepen.io/flaviocopes/pen/ZdWQPm

Color picker

You can let users pick a color using the `type="color"` element:

```
<input type="color" name="car-color">
```

You set a default value using the `value` attribute:

```
<input type="color" name="car-color" value="#000000">
```

The browser will take care of showing a color picker to the user.

Range

This input element shows a slider element. People can use it to move from a starting value to an ending value:

```
<input type="range" name="age" min="0" max="100" value="30">
```

You can provide an optional step:

```
<input type="range" name="age" min="0" max="100" value="30"
    step=
```

```
"10">
```

Telephone

The `type="tel"` input field is used to enter a phone number:

```
<input type="tel" name="telephone-number">
```

The main selling point for using `tel` over `text` is on mobile, where the device can choose to show a numeric keyboard.

Specify a `pattern` attribute for additional validation:

```
<input type="tel" pattern="[0-9]{3}-[0-9]{8}" name="telephone-num ber">
```

URL

The `type="url"` field is used to enter a URL.

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

You can validate it using the `pattern` attribute:

```
<input type="url" name="website" pattern="https://.*">
```

The `textarea` tag

The `textarea` element allows users to enter multi-line text. Compared to `input`, it requires an ending tag:

```
<textarea></textarea>
```

You can set the dimensions using CSS, but also using the `rows` and `cols` attributes:

```
<textarea rows="20" cols="10"></textarea>
```

As with the other form tags, the `name` attribute determines the name in the data sent to the server:

```
<textarea name="article"></textarea>
```

The `select` tag

This tag is used to create a drop-down menu.

The user can choose one of the options available.

Each option is created using the `option` tag. You add a name to the select, and a value to each option:

```
<select name="color">
    <option value="red">Red</option>
    <option value="yellow">Yellow</option>
</select>
```

You can set an option disabled:

```
<select name="color">
    <option value="red" disabled>Red</option>
    <option value="yellow">Yellow</option>
</select>
```

You can have one empty option:

```
<select name="color">
    <option value="">None</option>
    <option value="red">Red</option>
    <option value="yellow">Yellow</option>
</select>
```

Options can be grouped using the `optgroup` tag. Each option group has a `label` attribute:

```
<select name="color">
    <optgroup label="Primary">
        <option value="red">Red</option>
        <option value="yellow">Yellow</option>
        <option value="blue">Blue</option>
    </optgroup>
    <optgroup label="Others">
```

```
                                        <option value="green">Green</option>
                                        <option
    value="pink">Pink</option>
                                </optgroup>
</select>
```

Tables

Tables

In the early days of the web tables were a very important part of building layouts.

Later on they were replaced by CSS and its layout capabilities, and today we have powerful tools like CSS Flexbox and CSS Grid to build layouts. Tables are now used just for, guess what, building tables!

The `table` tag

You define a table using the `table` tag:

```
<table>

</table>
```

Inside the table we'll define the data. We reason in terms of rows, which means we add rows into a table (not columns). We'll define columns inside a row.

Rows

A row is added using the `tr` tag, and that's the only thing we can add into a `table` element:

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

This is a table with 3 rows.

The first row *can* take the role of the header.

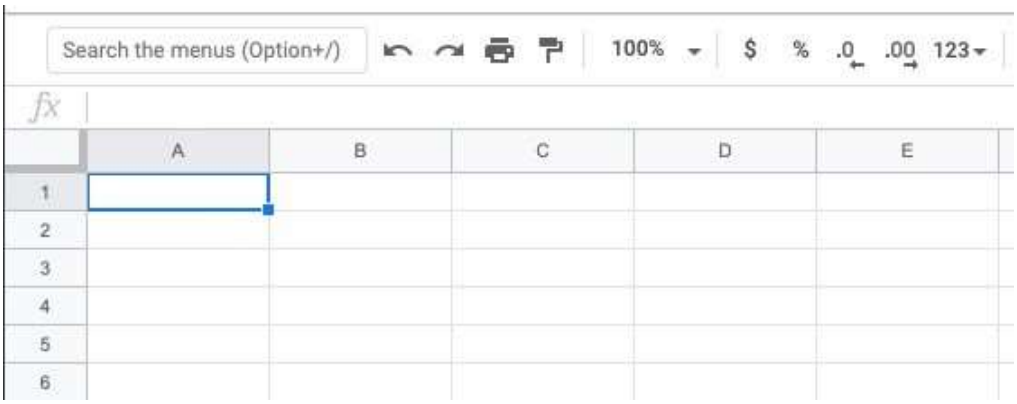
Column headers

The table header contains the name of a column, typically in a bold font.

Think about an Excel / Google Sheets document. The top

```
A-B-C-D...
```

header.



We define the header using the `th` tag:

```
<table>
    <tr>
        <th>Column 1</th>
        <th>Column 2</th>
        <th>Column 3</th>
    </tr>
    <tr></tr>
    <tr></tr>
</table>
```

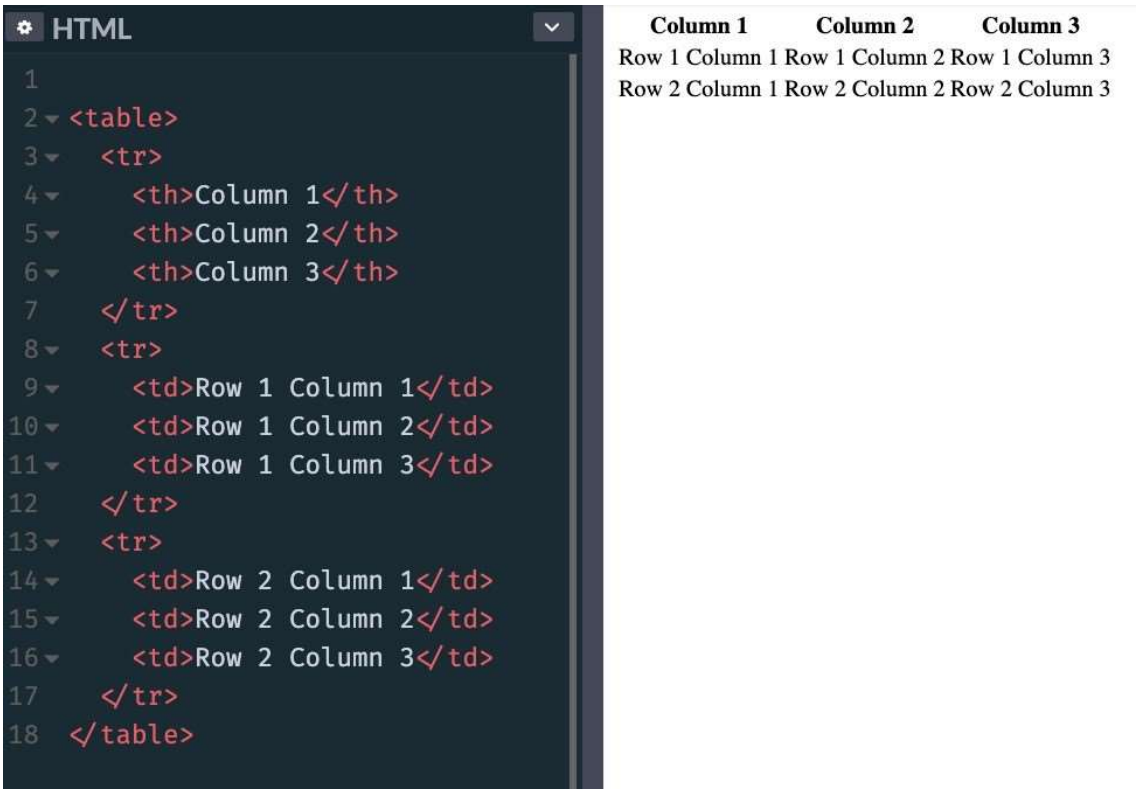
The table content

The content of the table is defined using `td` tags, inside the other `tr` elements:

```
<table>
    <tr>
        <th>Column 1</th>
        <th>Column 2</th>
        <th>Column 3</th>
    </tr>
    <tr>
        <td>Row 1 Column 1</td>
        <td>Row 1 Column 2</td>
    <td>Row 1 Column 3</td>
```

```
    </tr>
    <tr>
                <td>Row 2 Column 1</td>
                <td>Row 2 Column 2</td>
    <td>Row 2 Column 3</td>
    </tr>
</table>
```

This is how browsers render it, if you don't add any CSS styling:



Adding this CSS:

```
th, td {
    padding: 10px;
    border: 1px solid #333; }
```

makes the table look more like a proper table:

Column 1	Column 2	Column 3
Row 1 Column 1	Row 1 Column 2	Row 1 Column 3
Row 2 Column 1	Row 2 Column 2	Row 2 Column 3

Span columns androws

A row can decide to span over 2 or more columns, using the `colspan` attribute:

```
<table>
    <tr>
        <th>Column 1</th>
        <th>Column 2</th>
        <th>Column 3</th>
    </tr>
    <tr>
        <td colspan="2">Row 1 Columns 1-2</td>
        <td>Row 1 Column 3</td>
    </tr>
    <tr>
        <td colspan="3">Row 2 Columns 1-3</td>
    </tr>
</table>
```

Column 1	Column 2	Column 3
Row 1 Columns 1-2		Row 1 Column 3
Row 2 Columns 1-3		

Or it can span over 2 or more rows, using the `rowspan` attribute:

```
<table>
    <tr>
        <th>Column 1</th>
        <th>Column 2</th>
        <th>Column 3</th>
    </tr>
    <tr>
        <td colspan="2" rowspan="2">Rows 1-2 Columns 1-2</td>
        <td>Row 1 Column 3</td>
    </tr>
    <tr>
        <td>Row 2 Column 3</td>
    </tr>
</table>
```

Column 1	Column 2	Column 3
Rows 1-2 Columns 1-2		Row 1 Column 3
		Row 2 Column 3

Row headings

Before I explained how you can have column headings, using the `th` tag inside the first `tr` tag of the table.

You can add a `th` tag as the first element inside a `tr` that's not the first `tr` of the table, to have row headings:

```
<table>
    <tr>
        <th></th>
        <th>Column 2</th>
        <th>Column 3</th>
    </tr>
    <tr>
        <th>Row 1</th>
        <td>Col 2</td>
        <td>Col 3</td>
    </tr>
    <tr>
        <th>Row 2</th>
        <td>Col 2</td>
        <td>Col 3</td>
    </tr>
</table>
```

	Column 2	Column 3
Row 1	Col 2	Col 3
Row 1	Col 2	Col 3

More tags to organize the table

You can add 3 more tags into a table, to have it more organized.

This is best when using big tables. And to properly define a header and a footer, too.

Those tags are

- thead
- tbody
- tfoot

They wrap the tr tags to clearly define the different sections of the table. Here's an example:

```
<table>
    <thead>
                <tr>
                            <th></th>
                            <th>Column    2</th>
    <th>Column    3</th>
                </tr>
    </thead>
    <tbody>
                <tr>
                                        <th>Row       1</th>

            <td>Col 2</td>

        <td>Col 3</td>
                        </tr>
                        <tr>
                                        <th>Row       2</th>

            <td>Col 2</td>

        <td>Col 3</td>
                        </tr>
            </tbody>
            <tfoot>
                        <tr>
                                    <td></td>

        <td>Footer    of    Col    1</td>
```

```
                    <td>Footer      of
        Col     2</td>
                    </tr>
                </tfoot>
</table>
```

	Column 2	Column 3
Row 1	Col 2	Col 3
Row 2	Col 2	Col 3
	Footer of Col 1	Footer of Col 2

Table caption

A table should have a `caption` tag that describes its content. That tag should be put immediately after the opening `table` tag:

```
<table>
    <caption>Dogs  age</caption>
```

```
        <tr>
            <th>Dog</th>
            <th>Age</th>
        </tr>
        <tr>
            <td>Roger</td>
            <td>7</td>
        </tr>
</table>
```

Multimedia tags: audio and video

Multimedia tags: `audio` and `video`

In this section I want to show you the `audio` and `video` tags.

The `audio` tag

This tag allows you to embed audio content in your HTML pages.

This element can stream audio, maybe using a microphone via `getUserMedia()`, or it can play an audio source which you reference using the `src` attribute:

```
<audio src="file.mp3">
```

By default the browser does not show any controls for this element. Which means the audio will play only if

set to autoplay (more on this later) and the user can't see how to stop it or control the volume or movethrough the track.

To show the built-in controls, you can add the `controls` attribute:

```
<audio src="file.mp3" controls>
```

Controls can have a custom skin.

You can specify the MIME type of the audiofile using the `type` attribute. If not set, the browser will try to automatically determine it:

```
<audio src="file.mp3" controls type="audio/mpeg">
```

An audiofile by default does not play automatically. Add the `autoplay` attribute to play the audioautomatically:

```
<audio src="file.mp3" controls autoplay>
```

> Note: mobile browsers don't allow autoplay

The `loop` attribute restarts the audioplaying at 0:00 if set; otherwise, if not present, the audiostopsat the end of the file:

```
<audio src="file.mp3" controls autoplay loop>
```

You can also play an audio file muted using the `muted` attribute (not really sure what's the usefulness of this):

```
<audio src="file.mp3" controls autoplay loop muted>
```

Using JavaScript you can listen for various events happening on an `audio` element, the most basic of which are:

- `play`: when the file starts playing
- `pause`: when the audio playing was paused
- `playing`: when the audio is resumed from a pause
- `ended`: when the end of the audio file was reached

The `video` tag

This tag allows you to embed video content in your HTML pages.

This element can stream video, using a webcam via `getUserMedia()` or **WebRTC**, or it can play a video source which you reference using the `src` attribute:

```
<video src="file.mp4">
```

By default the browser does not show any controls for this element, just the video.

Which means the video will play only if set to autoplay (more on this later) and the user can't see how to stop it, pause it, control the volume or skip to a specific position in the video.

To show the built-in controls, you can add the `controls` attribute:

```
<video src="file.mp4" controls>
```

Controls can have a custom skin.

You can specify the MIME type of the video file using the `type` attribute. If not set, the browser will try to automatically determine it:

```
<video src="file.mp4" controls type="video/mp4">
```

A video file by default does not play automatically. Add the `autoplay` attribute to play the video automatically:

```
<video src="file.mp4" controls autoplay>
```

Some browsers also require the `muted` attribute to autoplay. The video autoplays only if muted:

```
<audio src="file.mp3" controls autoplay muted>
```

The `loop` attribute restarts the video playing at 0:00 if set; otherwise, if not present, the video stops at the end of the file:

```
<video src="file.mp4" controls autoplay loop>
```

You can set an image to be the poster image:

```
<video src="file.mp4" poster="picture.png">
```

If not present, the browser will display the first frame of the video as soon as it's available.

You can set the `width` and `height` attributes to set the space that the element will take so that the browser can account for it and it does not change the layout when it's finally loaded. It takes a numeric value, expressed in pixels.

Using JavaScript you can listen for various events happening on an `video` element, the most basic of which are:

- `play` when the file starts playing
- `pause` when the video was paused
- `playing` when the video is resumed from a pause
- `ended` when the end of the video file was reached

iframes

iframes

The `iframe` tag allows us to embed content coming from other origins (other sites) into our web page.

Technically, an iframe creates a new nested browsing context. This means that anything in the iframe does not interfere with the parent page, and vice versa. JavaScript and CSS do not "leak" to/from iframes.

Many sites use iframes to perform various things. You might be familiar with Codepen, Glitch or other sites that allow you to code in one part of the page, and you see the result in a box. That's an iframe.

You create one this way:

```
<iframe src="page.html"></iframe>
```

You can load an absolute URL, too:

```
<iframe src="https://site.com/page.html"></iframe>
```

You can set a set of width and height parameters (or set them using

CSS) otherwise the iframe will use the defaults, a 300x150 pixels box:

```
<iframe src="page.html" width="800" height="400"></iframe>
```

Srcdoc

The `srcdoc` attribute lets you specify some inline HTML to show. It's an alternative to `src`, but recent and not supported in Edge 18 and lower, and in IE:

```
<iframe srcdoc="<p>My dog is a good dog</p>"></iframe>
```

Sandbox

The `sandbox` attribute allows us to limit the operations allowed in the iframes.

If we omit it, everything is allowed:

```
<iframe src="page.html"></iframe>
```

If we set it to "", nothing is allowed:

```
<iframe src="page.html" sandbox=""></iframe>
```

We can select what to allow by adding options in the `sandbox` attribute. You can allow multiple ones by adding a space in between.

Here's an incomplete list of the options you can use:

- `allow-forms`: allow to submit forms
- `allow-modals` allow to open modals windows, including calling `alert()` in JavaScript
- `allow-orientation-lock` allow to lock the screen orientation
- `allow-popups` allow popups, using `window.open()` and `target="_blank"` links
- `allow-same-origin` treat the resource being loaded as same origin
- `allow-scripts` lets the loaded iframe run scripts (but not create popups).
- `allow-top-navigation` gives access to the iframe to the top level browsing context

Allow

Currently experimental and only supported by Chromium-based browsers, this is the future of resource sharing between the parent window and the iframe.

It's similar to the `sandbox` attribute, but lets us allow specific features, including:

- `accelerometer` gives access to the Sensors API Accelerometer interface
- `ambient-light-sensor` gives access to the Sensors API AmbientLightSensor interface
- `autoplay` allows to autoplay video and audiofiles
- `camera`

allows to access the camera from the getUserMedia API `display-capture` allows to access the screen content using the getDisplayMedia API `fullscreen` allows to access

- fullscreen mode `geolocation` allows to access
- the Geolocation API `gyroscope` gives access
- to the Sensors API Gyroscope interface `magnetometer`
- gives access to the Sensors API Magnetometer interface • `microphone` gives access to the device microphone using the getUserMedia API `midi` allows access to
- the Web MIDI API `payment` gives access to
- the Payment Request API `speaker` allows
- access to playing audio through the device speakers `usb` gives access to
- the WebUSB API. `vibrate` gives
- access to the Vibration API `vr`
- gives access to the WebVR API

Referrer

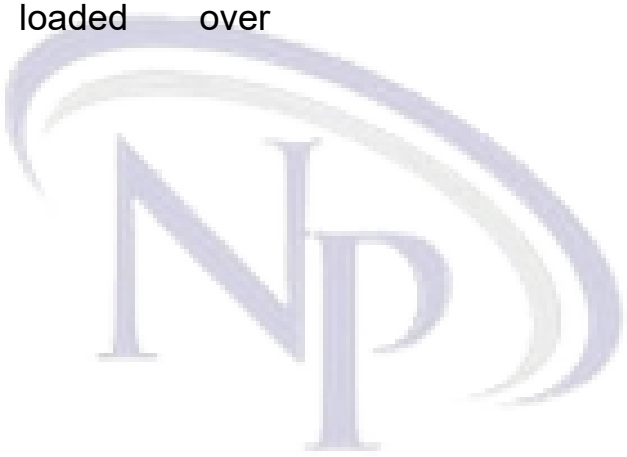
When loading an iframe, the browser sends it important information about who is loading it in the `Referer` header (notice the single `r`, a typo we must live with).

> The misspelling of referrer originated in the original proposal by computer scientist Phillip Hallam-Baker to incorporate the field into the HTTP specification. The misspelling was set in stone by the time of its incorporation into the Request for Comments standards document RFC 1945

The `referrerpolicy` attribute lets us set the referrer to send to the iframe when loading it. The referrer is an HTTP header that lets the page know who is loading it. These are the allowed values:

- `no-referrer-when-downgrade` it's the default, and does not send the referrer when the current page is loaded over HTTPS and the iframe loads on the HTTP protocol
- `no-referrer` does not send the referrer header
- `origin` the referrer is sent, and only contains the origin (port, protocol, domain), not the origin + path which is the default
- `origin-when-cross-origin` when loading from the same origin (port, protocol, domain) in the iframe, the referrer is sent in its complete form (origin + path). Otherwise only the origin is sent
- `same-origin` the referrer is sent only when loading from the same origin (port, protocol, domain) in the iframe
- `strict-origin` sends the origin as the referrer if the current page

is loaded over HTTPS and the iframe also loads on the HTTPS protocol. Sends nothing if the iframe is loaded over HTTP.
- `strict-origin-when-cross-origin` sends the origin+path as the referrer when working on the same origin. Sends the origin as the referrer if the current page is loaded over HTTPS and the iframe also loads on the HTTPS protocol. Sends nothing if the iframe is loaded over HTTP.
- `unsafe-url`: sends the origin+path as the referrer even when loading resources from HTTP and the current page is loaded over HTTPS



Images

Images

Images can be displayed using the `img` tag.

This tag accepts a `src` attribute, which we use to set the image source:

```
<imgsrc="image.png">
```

We can use a wide set of images. The most common ones are PNG, JPEG, GIF, SVG and more recently WebP.

The HTML standard requires an `alt` attribute to be present, to describe the image. This is used by screen readers and also by search engine bots:

```
<imgsrc="dog.png" alt="A picture of a dog">
```

You can set the `width` and `height` attributes to set the space that the element will take, so that the browser can account for it and it does not change the layout when it's fully loaded. It takes a numeric value, expressed in pixels.

```
<img src="dog.png" alt="A picture of a dog" width="300" height="2
```

```
00">
```

The `figure` tag

The `figure` tag is often used alongwith the `img` tag. `figure` is a semantic tag often used when you want to display an image with a caption. You use it like this:

```
<figure>
    <img src="dog.png"
         alt="A nice dog">
    <figcaption>A nice dog</figcaption>
</figure>
```

The `figcaption` tag wraps the caption text.

Responsive images using `srcset`

The `srcset` attribute allows you to set responsive images that the browser can use depending on the pixel density or window width, according to your preferences. This way, it can only download the resources it needs to render the page, without downloading a bigger image if it's on a mobile device, for example.

Here's an example, where we give 4 additional images for 4 different screen sizes:

```
<imgsrc="dog.png"
    alt="A picture of a dog"
                srcset="dog-500.png
    500w,

                    dog-800.png
    800w,

          dog-1000.png 1000w,

                        dog-
1400.png  1400w">
```

In the `srcset` we use the `w` measure to indicate the window width.

Since we do so, we also need to use the `sizes` attribute:

```
<img  src="dog.png"
      alt="A picture of a
      dog"
                    sizes="(max-width: 500px) 100vw, (max-width:
      900px) 50vw,  800 px"
                    srcset="dog-500.png
      500w,

            dog-800.png  800w,

                  dog-1000.png 1000w,
```

```
dog-1400.png  1400w">
```

In this example the `(max-width: 500px) 100vw, (max-width: 900px) 50vw, 800px` string in the `sizes` attribute describes the size of the image in relation to the viewport, with multiple conditions separated by a semicolon.

The media condition `max-width: 500px` sets the size of the image in correlation to the viewport width. In short, if the window size is < 500px, it renders the image at 100% of the window size.

If the window size is bigger but < `900px`, it renders the image at 50% of the window size.

And if even bigger, it renders the image at 800px.

The `vw` unit of measure can be new to you, and in short we can say that 1 `vw` is 1% of the window width, so `100vw` is 100% of the window width.

A useful website to generate the `srcset` and progressively smaller images is https://responsivebreakpoints.com/.

The `picture` tag

HTML also gives us the `picture` tag, which does a very similar job to `srcset`, and the differences are very subtle.

You use `picture` when instead of just serving a smaller version of a file, you completely want to change it. Or serve a different image format.

The best use case I found is when serving a WebP image, which is a format still not widely supported. In the `picture` tag you specify a list of images, and they will be used in order, so in the next example, browsers that support WebP will use the first image, and fallback to JPG if not:

```
<picture>
    <source type="image/webp" srcset="image.webp">
    <img src="image.jpg" alt="An image">
</picture>
```

> The `source` tag defines one (or more) formats for the images. The `img` tag is the fallback in case the browser is very old and does not support the `picture` tag.

In the `source` tag inside `picture` you can add a `media` attribute to setmedia queries.

The example that follows kind of works like the above example with `srcset`:

```
<picture>
    <sourcemedia="(min-width: 500w)" srcset="dog-500.png" sizes="100vw">
    <sourcemedia="(min-width: 800w)" srcset="dog-800.png" sizes="100vw">
    <sourcemedia="(min-width: 1000w)" srcset="dog-1000.png" siz es="800px">
    <sourcemedia="(min-width: 1400w)" srcset="dog-1400.png" siz es="800px">
    <img src="dog.png" alt="A dog image">
</picture>
```

But that's not its use case, because as you can see it's muchmore verbose.

The `picture` tag is recent but is now supported by all the major browsers except Opera Mini and IE (all versions).

Accessibility

Accessibility

It's important we design our HTML with accessibility in mind.

Having accessible HTML means that people with disabilities can use the Web. There are totally blind or visually impaired users, people with hearing loss issues and a multitude of other different disabilities.

Unfortunately this topic does not take the importanceit needs, and it doesn't seemas cool as others.

What if a person can't *see* your page, but still wants to consume its content? First, how do they do that? They can't use the mouse, they use something called a **screen reader**. You don't have to imagine that. You can try one now: Google provides the free ChromeVox Chrome Extension. Accessibility must also take care of allowing tools to easily select elements or navigate through the pages.

Web pages and Web apps are not always built with accessibility as one of their first goals, and maybe version 1 is released not accessible but it's possible to makea web page accessible after the fact. Sooner is better, but it's never too late.

It's important and in my country, websites built by the government or other public organizations must be accessible.

What does this mean to makean HTML accessible? Let me illustrate the main things you need to think about.

> Note: there are several other things to take care about, which might go in the CSS topic, like colors, contrast and fonts. Or how to make SVG images accessible. I don't talk about them here.

Use semantic HTML

Semantic HTML is very important and it's one of the main things you need to take care of. Let me illustrate a few common scenarios.

It's important to use the correct structure for heading tags. The most important is `h1`, and you use higher numbers for less important ones, but all the same-level headings should have the samemeaning (think about it like a tree structure)

```
h1
        h2
                h3
        h2
```

```
h2

h3

h4
```

Use `strong` and `em` instead of `b` and `i`. Visually they look the same, but the first 2 have more meaning associated with them. `b` and `i` are more visual elements.

Lists are important. A screen reader can detect a list and provide an overview, then let the user choose to get into the list or not.

A table should have a `caption` tag that describes its content:

```
<table>
    <caption>Dogs age</caption>
    <tr>
        <th>Dog</th>
        <th>Age</th>
    </tr>
    <tr>
        <td>Roger</td>
        <td>7</td>
```

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

Use `alt` attributes for images

All images must have an `alt` tag describing the image content. It's not just a good practice, it's required by the HTML standard and your HTMLwithout it is not validated.

```
<imgsrc="dog.png" alt="picture of my dog">
```

It's also good for search engines, if that's an incentive for you to add it.

Use the `role` attribute

The `role` attribute lets you assign specific roles to the various elements in your page.

You can assign lots of different roles:complementary, list, listitem, main, navigation, region, tab, alert, application, article, banner, button, cell, checkbox, contentinfo, dialog, document, feed, figure, form, grid, gridcell, heading, img, listbox, row, rowgroup, search, switch, table, tabpanel, textbox, timer.

It's a lot and for the full reference of each of them I give you this MDN link. But you don't need to assign a role to everyelement in the page. Screen readers can infer from the HTML tag in most cases. For example you don't need to add a role tag to semantic tags like `nav`, `button`, `form`.

Let's take the `nav` tag example. You can use it to define the page navigation like this:

```
<nav>
    <ul>
                <li><a href="/">Home</a></li>                <li><a
    href="/blog">Blog</a></li>
    </ul>
</nav>
```

If you were *forced* to use a `div` tag instead of `nav`, you'd use the `navigation` role:

```
<div role="navigation">
    <ul>
                <li><a href="/">Home</a></li>                <li><a
    href="/blog">Blog</a></li>
    </ul>
</div>
```

So here you got a practical example: `role` is used to assign a meaningful value when the tag does not convey the meaning already.

Use the `tabindex` attribute

The `tabindex` attribute allows you to change the order of how pressing the Tab key selects "selectable" elements. By defaults only links and form elements are "selectable" by navigation using the Tab key (and you don't need to set `tabindex` on them).

Adding `tabindex="0"` makes an element selectable:

```
<div tabindex="0">...
</div>
```

Using `tabindex="-1"` instead removes an element from this tab-based navigation, and it can be pretty useful.

Use the `aria` attributes

ARIA is an acronym that means Accessible Rich Internet Applications and defines semantics that can be applied to elements.

`aria-label`

This attribute is used to add a stringto describe an element.

Example:

```
<p aria-label="The description of the product">...</p>
```

I use this attribute on my blog sidebar, where I have an input box for search without an explicit label, as it has a placeholder attribute.

aria-labelledby

This attribute sets a correlation between the current element and the one that labels it.

If you know how an `input` element can be associated to a `label` element, that's similar.

We pass the item id that describes the current element.

Example:

```
<h3 id="description">The description of the product</h3>

<p aria-labelledby="description"> ...
</p>
```

aria-describedby

This attribute lets us associate an element with another element that serves as description.

Example:

```
<button aria-describedby="payNowDescription" >Pay now</button>

<div id="payNowDescription">Clicking the button will send you to
    our Stripe form!</div>
```

Usearia-hiddento hide content

I like a minimalistic design in my sites. My blog for example is mostly just content, with somelinks in the sidebar. But somethings in the sidebarare just visual elements that don't add up to the experienceof a person that can't see the page. Like my logo picture, or the dark/bright theme selector.

Adding the `aria-hidden="true"` attribute will tell screen readers to ignore that element.

Where to learn more

This is just an introduction to the topic. To learn more, I recommend these resources:

- https://www.w3.org/TR/WCAG20/ https://webaim.org
-
- https://developers.google.com/web/fundamentals/accessibility/