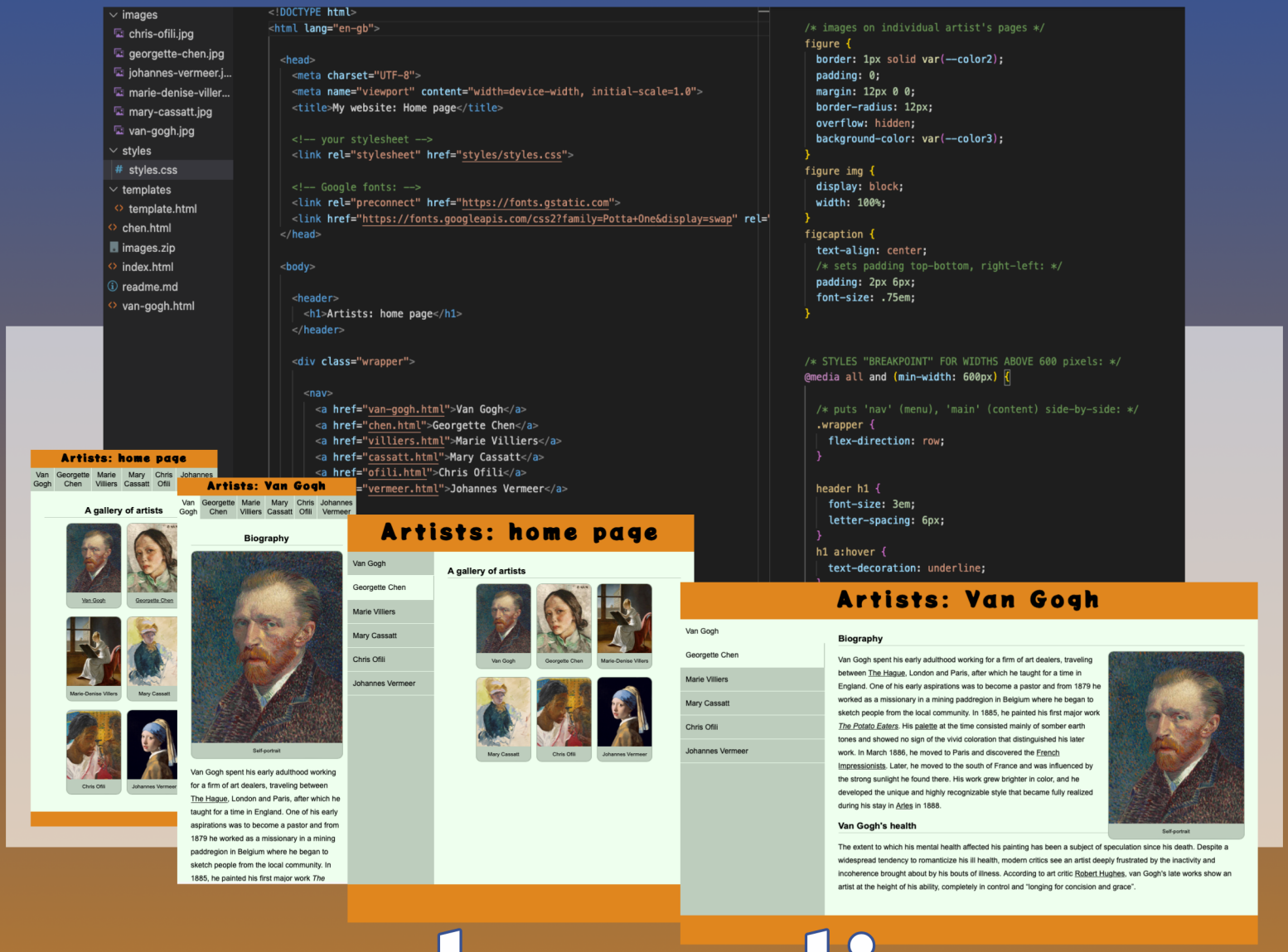
Second Edition 2022



web coding first steps

a simple introduction to responsive web design for mobile, tablet, laptop and desktop screens

Dave Everitt

Web Coding First Steps

Second Edition

Dave Everitt

This book is for sale at http://leanpub.com/webcodingfirststeps

This version was published on 2022-10-09

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

What you will learn

Simple design and code

This compact book is for beginners to HTML and CSS who want to build a simple website, or need to understand these languages to create or modify existing templates, themes, etc. You will learn the essential parts of modern HTML5 and CSS code, using free resources like the Atom[1] or VSCode[2] programmer's text editor and your web browser (Chrome is recommended[3] for this tutorial, but any up-to-date browser is also fine).

As well as learning the basics of these markup languages, another aim is to help you understand how to *design* a website. Learning the underlying code will reveal the kinds of constraints and opportunities the web brings to the design process. You will find out how to use a few of the most practical HTML5 tags (there are many others you can ignore and will probably never need), and to use modern CSS to style their appearance.

HTML and CSS

HTML (HyperText Markup Language) and CSS (Cascading Style Sheets) are *markup* languages, they do not contain any of the complex *logic* found in the *programming* languages you may have heard about (e.g. Python, JavaScript, PHP, Ruby…). Markup languages are mainly used to transform human-readable text and other media into computer-readable material. For instance, CSS describes styles like measurements, colours, fonts, etc. for computers to interpret and make visible in a web browser. The specifications for these languages are governed by groups of industry specialists so that—in the case of HTML and CSS—the manufacturers of browsers such as Chrome, Firefox or Microsoft Edge have common guidelines to follow, or can contribute future changes.

'HTML' stands for what it is and does: *HyperText Markup Language*—'Hypertext' here means text that contains rich content such as links and other features unavailable in ordinary text. CSS stands for *Cascading Style Sheets*, 'Style Sheets' is obvious, but the meaning of 'Cascading' will become apparent later in this book.

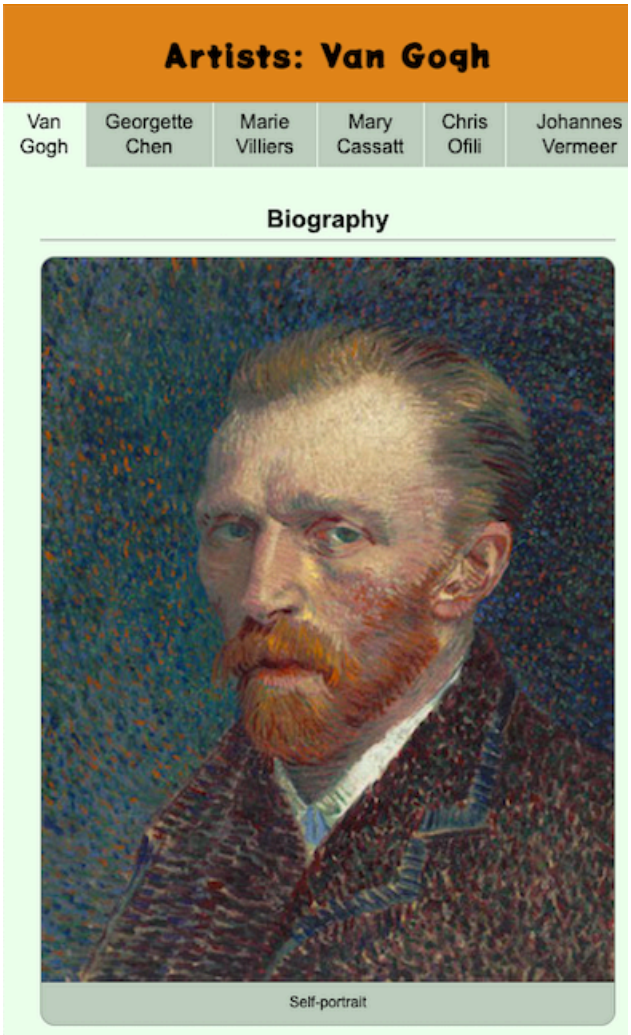
[1] https://atom.io/
[2] https://code.visualstudio.com
[3] https://www.google.co.uk/chrome/browser/desktop/

What you're going to build

To help you grasp how design and code relate to each other, we're going to build something like this page:

Picture 1: the final result

Another thing we'll be doing is making our website *responsive*, this means that the website will respond to the screen that it is being displayed on and restructure itself accordingly. So, when viewed on a mobile device the website we build will look like *Picture 2*.



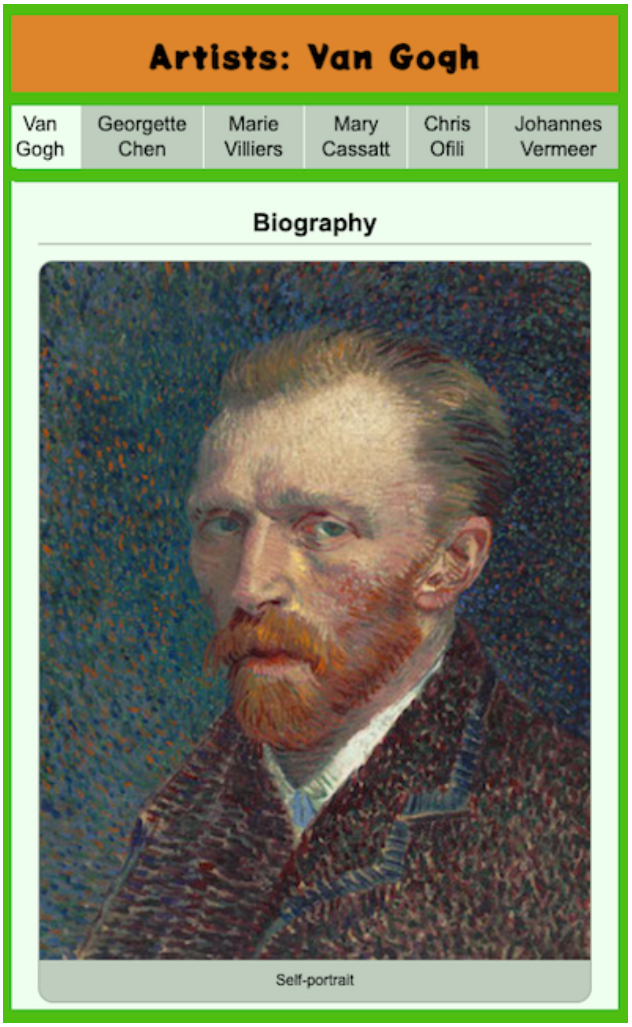
Picture 2: the final result on mobile browser

There are 4 main boxes (it looks like 3 in the screenshot in *Picture 3* because one of the boxes surrounds and contains *two* of the others). By styling these box elements individually, the page is created and designed. It is important when thinking about designs for web pages to make sure they are easy—or even possible—to build using HTML, so it is important to grasp that **everything on a web page starts with a box**, even if the original box is invisible (unless you add a border, of course), or if you choose to break out of the boundaries of the containing box, say with some CSS styles that move an image from its default position.

So for an easy life when designing a website, think BOX! In *Picture 3* you can see all the main boxes outlined in green.

Picture 3: the final result outlined

One of the reasons this type of layout is still so common on the web (header, left or right sidebar column with a menu, main content area) is that it is easy to achieve and has become familiar to internet users. Even though many modern designs may look different, the underlying code is the same. Also by using these three main boxes you can rearrange them easily when they're being viewed in different contexts (e.g. desktop, tablet or mobile). Here's an image of the same page with the same boxes outlined but viewed on a mobile browser. Notice how the boxes have been rearranged to suit the narrower screen size.



Picture 4: the final result outlined on a mobile browser

Your creativity can show when you learn the basics and realise that your site doesn't have to look like any other and, at the same time, you also understand that visitors to the site need to be able to find and browse content without getting frustrated, lost or confused!

The difference between *static* and *dynamic* websites

The example site you build here will contain one `.html` file for each page. It will be a *static website* i.e. one that doesn't change according to a website visitor's input or preferences. To update a *static* site you edit the necessary files and upload them to a *web server*, updating things by replacing the old ones. Some websites and blogs are created by static website generators[4], which can be programmed to automatically output and upload the same kind of *static* pages you'll be building. However, none of the many current static website generators[5] are covered in this version of the book, because as well

[4] http://davidwalsh.name/introduction-static-site-generators
[5] https://staticsitegenerators.net/

as the *markup languages* HTML and CSS, they involve learning *programming languages*. However, you *will* be able to use your HTML and CSS skills if you use those tools in future.

Most of the websites you are probably familiar with (Facebook, Amazon, Pinterest, etc.) are *dynamic websites*, created from various components and assembled in response to searches, user input and preferences. They can be constructed using many different *programming* languages, and most will read from and write to *databases*, so that—for instance—user choices, user uploads and comments persist between different pages and visits on different days; or fresh content is 'pushed' to your browser as it is created. The skills needed for this kind of functionality are not trivial, and far harder to learn than HTML and CSS. Although content management tools like WordPress can make this process easier, they may also get in the way if you want to do something different or unique.

In the end, no matter how a site is made, the result is always the same: a web page that appears in a web browser, made from code. This is why HTML5 and CSS are essential tools in any web designer and developer's skillset.

Setting things up

This is often called 'setting up your development environment', and involves some simple housekeeping tasks that will make it easier to get things done and find your way around later.

The code (text) editor

If it is not already on your computer, download the latest version of Atom from the website[6], install and launch it (as you would any other application). If you don't have admin rights on your computer, you can move and run Atom from—for example—the documents folder or even the desktop.

If you don't see the pane that will contain your files (the 'tree view') you can use `cmd-\`

A programmers' text editor like Atom (or VSCode[7]) is more suited to writing code than Apple's *TextEdit* or Windows' *Notepad* (you can't use Word or other word processors for writing code). For now, it's enough to know that it highlights the various parts of the code in different colours so you can find your way around easily, and has a side menu for accessing all your website files.

If you want to learn how to use keyboard shortcuts for your text editor (recommended!), there are handy cheat sheets for Atom on Apple[8] and Windows[9] (to use the more detailed Windows cheat sheet for Apple, substitute `cmd` for shortcuts starting with `ctrl`). There's a VSCode cheetsheet here[10].

Make a folder for your website

This can be anywhere you like, but it's best to have your project folders in one location. Call it something to reflect the name of your website—it doesn't matter what, but:

1. keep it brief and all lower-case, no capital letters
2. don't use spaces, divide words with hyphens (the minus sign) or underscores.

Let's say it's called 'my-site'. Inside the 'my-site' folder, add three more folders: 'javascript', 'images' and 'styles'. This helps to keep things organised (you won't be using Javascript in this tutorial but the folder will be there in case you decide to experiment with it later on):

[6] https://atom.io
[7] https://code.visualstudio.com
[8] https://bugsnag.com/blog/atom-editor-cheat-sheet
[9] http://www.shortcutworld.com/en/win/Atom-(text-editor)_1.0.html
[10] https://www.crio.do/blog/vs-code-shortcuts/

```
my-site/
  images/
  javascript/
  styles/
```

Before you start, drag the folder `my-site` to the Atom icon to open a sidebar in Atom showing all the folders. Wait for the list of your folders to show up—it may take a few seconds.

Atom, like most programmers' text editors, has a sidebar showing the files you're working on (this is the 'Tree View' at the left of the main editing pane—if you can't see it, hit `cmd-\` or `ctrl-\` (Apple/Windows)—this will also hide it if it's showing. Here, you can duplicate and create folders and files, drag files in and out of folders, etc. If you right-click (or ctrl-click) in the sidebar, either in an empty space or on a file you want to duplicate, you will see a *contextual menu* with actions you can take.

Create your first HTML5 page

in Atom, create a new file (`right-click` or `ctrl-click` in the 'tree view' sidebar) and save it with the filename 'index.html' in your 'my-site' folder.

Then create another new file with the filename 'styles.css' in the 'styles' folder inside your 'my-site' folder. The 'styles.css' file will contain your *stylesheet* code—i.e. all the styling information for your HTML pages.

Your website folder should now contain the following (if you've saved files in the wrong place, drag them into the right place in the Atom sidebar (if you do this in the Mac OS X Finder or Windows Explorer, Atom will update the tree view automatically):

```
my-site/
  index.html
  images/
  javascript/
  styles/
    styles.css
```

The Atom sidebar will help you keep track of all your website files, but also keep the 'my-site' folder open on your desktop so you can double-check where your files are if necessary.

In 'index.html' add the following HTML code, taking care to type it *exactly* as shown (you can copy-paste, but check carefully that your pasted code looks just the same):

```
<!DOCTYPE html>
<html lang="en-gb">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My website: Home page</title>
  </head>
  <body>
    my webpage content
  </body>
</html>
```

The `<head>` and `<body>` sections

In the code above, the `<html>` tag contains your entire HTML document. Inside it are two main areas: `<head>` and `<body>`, both with matching *closing* tags—*every site on the web* has these tags!

The `<head>` section contains information the browser needs to create your HTML page, but *does not appear* in the browser window. So far, it contains the following:

Two `<meta>` tags (these never need changing—you can ignore them):

`UTF-8` means you can write text with accents, typographic quotes and dashes, and other characters.

`name="viewport" content="width (…)` is the basic information needed for mobile devices to display the web page.

`<title>` tag:

The name or description of your HTML page that appear at the top of some browser windows; search engines also read this to help people find your page.

The `<body>` section of your web page will contain all the *visible page content* that appears in a browser. Visible content goes between the opening HTML `<body>` and closing `</body>` tags.

In HTML there is usually an opening and a closing HTML 'tag' with the same name—the closing tag has a forward slash before the tag name. Other tags—like the opening `<div>` and closing `</div>` and the text between them, are 'nested' inside these. A few tags, like the `<meta…` tag in the `<head>` section above, or the `<img…` tag you will use later in this tutorial, don't need a closing tag.

The first step—text on a blank page!

Now you've made a change to your document, save the file. You can do this by using the shortcut `cmd-S`/`ctrl-S` (Apple/Windows).

Open your newly-saved 'index.html' file in a browser—a quick way is to drag it from the Finder to the browser icon or a blank browser window.

You should see a white page with the text 'my webpage content' in black. Don't close the browser window—leave it open, because you can refresh it each time you make a change (`cmd-R` or `crtl-R` Apple/Windows).

Boxes

Building web pages is all about putting content into boxes (HTML), then styling these boxes (CSS) to appear where you want them, and making the boxes and their content appear as you want. Boxes can have borders and backgrounds, and contain other boxes, text and multiple images. Text can be styled with typefaces, in any size or colour.

Before you begin a website design, it's good practice to make a rough drawing or plan, showing what boxes you want and where you want them to be on the page. It can be done quite quickly with a pencil and paper. You don't need to do this for the the tutorial, but will definitely want to sketch things out roughly once you start to design your own website.

There are certain HTML tags that create the boxes for your web page layout, the most basic is the HTML `<div>` tag), while `<header>`, `<main>`, `<nav>` and `<footer>` are called 'semantic' tags because they describe what they contain. You will use some of these later.

the current version of HTML (HTML5) provides *semantic* or meaningful tags for the 'boxes' that now make up page sections e.g. `<header>` instead of `<div id="header">`. Before HTML5, designers used to give their own `id` names names to `divs`, such as `heading`, `top`, `header_bar` or whatever they wanted, but (for example) the `<header>` tag is now part of the HTML standard. On most well-maintained sites, HTML5 tags have replaced older `div`-based HTML4 code. Search engines, text readers for visually-impaired users, and other services that machine-read HTML pages can now understand the *meaning* behind these tags.

Linking the stylesheet

Your 'index.html' page needs to be able to *find the stylesheet*. In the `<head>` section at the top of your page, add a `<link…` tag to the stylesheet:

```
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Artists: Home page</title>
    <link rel="stylesheet" href="styles/styles.css">
  </head>
```

This new `<link…` tag in the `<head>` section shows the browser where to find your CSS *stylesheet* in its `href` property (`styles/styles.css`)—`styles.css` is *inside* the `styles` folder in your website's containing or 'root' folder.

The 'root' folder contains the entire website and the `index.html` file is set by web servers as the *default* page. It appears when you type in a domain name, without you needing to type (for instance) `google.com/index.html`. Any folder inside a website can have an `index.html` file that acts as a similar default file for *that particular folder*. Some web programming software uses this convention to make simpler URLs, such as `mywebsite.com/about/`, rather than `mywebsite.com/about/index.html`. When launching a live website on a *web server* you can do the same, although paths to images and stylesheets etc. in `index.html` files that are *inside folders* will need a *forward slash* at the start to tell the browser to find files *from the 'website root'* (`/`) (e.g. `/styles/styles.css` or `/images/mylogo.png`).

Setting some CSS variables

Before adding some styles for the page, we're first going to set some variables or *custom properties* in the CSS to set the colours used for our website. Remembering and retyping the *hexadecimal* values for colours is difficult, so giving each colour a simple name makes using and changing them much easier.

Open the `styles.css` page in your editor and add the following code:

```
:root {
  --color1: #e0861e;
  --color2: #9a9;
  --color3: #bdcdbd;
  --color4: #efe;
}
```

In the above code we set the variables as `:root` variables which means they have a global scope and can be used throughout the rest of your CSS styesheet for the whole website. We give each of the colours a simple but memorable name such as `--color1`. Hexadecimal colour values are prefixed by a hash symbol `#`, followed by 3 or 6 characters. (For more information about the CSS colours see Appendix 1: CSS colours).

The hash (UK) or 'pound' (US) symbol `#` is `alt-3` or `shift-3` (Apple/Windows, depending on whether your language is set to UK or US English). If you can't find it, copy and paste this symbol: #).

Adding a style

Now we've set some variables let's add some styles to the `<body>` tag to change its appearance.

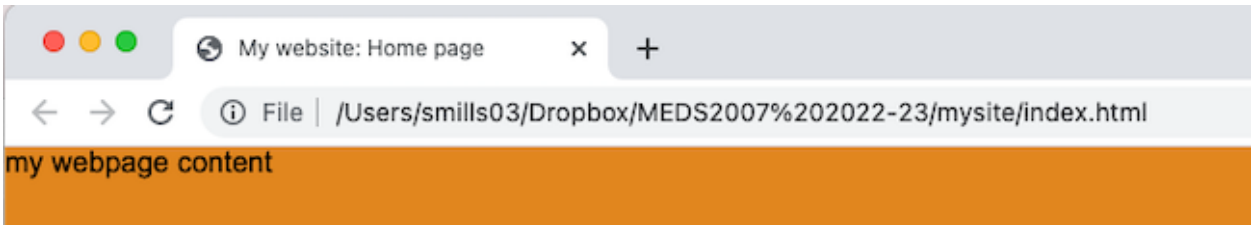
The `<body>` tag is the *mother of all boxes*, in which all other boxes are contained. It is displayed as the whole browser window. In most browsers it has some default margin or padding, so it is common practice to set these both to zero. This prevents any default blank space around the page.

In `styles.css`, the `background-color` style (note the spelling of 'color') uses one of the colours we just set to add colour to the page background, and a default `font-family` tag sets the defaul font for the page. Be sure to use a colon `:` and semicolon `;` in the right places:

```
body {
  margin: 0;
  padding: 0;
  background-color: var(--color1);
  font-family: Arial, Helvetica, sans-serif;
}
```

Don't worry about understanding this yet, although some parts might appear obvious, because they *do what they say*.

Save your page and reload it in your browser to see something like this:



Picture 1: your page should look like the above image

You can see that the background colour of the page has been changed to the orange defined by the `color1` variable we set earlier. It might look unimpressive so far, but it's a start!

Adding a header section

Now, we will use the HTML `<header>` tag to add a first box to the page.

Note: `<header>` **appears in the browser window**, and is **distinct from the** `<head>` **tag** — the invisible section the browser uses to set things up. So **do not** put this code in the `<head>` section!

In the `index.html` file delete the current text `my webpage content` and in its place, between the two `<body>` tags type in a new `<header>` tag, with the text 'Artists: home page' between the opening and closing `<header>` tags, like this:

```
<body>
  <header>
    Artists: home page
  </header>
</body>
```

Note how the `header` tag is *nested* as a *child* inside the *parent* `body` tag. We show this by **indenting with two spaces** inside the `<body>` tag. You will soon see why this is good practice, because we'll mention it a few times more.

Now you can style the `<header>` tag in `styles.css`, with styles that align the text centrally using `flex` (more about `flex` or *CSS flexbox* later).

```
header {
  display: flex;
  justify-content: center;
  align-items: center;
}
```

Note! the CSS selector `header` in our *CSS file* targets the HTML `<header>` tag in our *HTML file*. Although they're connected, don't confuse the two between your HTML (angle brackets: `< … >`) and CSS (curly brackets: `{ … }`) files—the two kinds of brackets are a giveaway.

Save your changes and reload the page in your web browser—you should see the words 'Artists: home page', centrally aligned both horizontally (`justify-content`) and vertically (`align-items`) although at this stage the `<body>` background colour underneath the `<header>` shows through and appears underneath it too, so you can't see the *vertical alignment* properly:

Picture 2: your page should look like the above image

With no CSS `height` rule, HTML 'block' elements (like `header` or `div` tags) *increase in height* to 'contain' the elements inside them. If you add a lot more text inside the `<header>` tag you will see this happen, but remove it afterwards!

Styling the page title

Every page needs a level one heading `<h1>` to describe its content. In `index.html`, hit return just before the 'A' of *Artists: home page*, then surround the words with opening and closing `<h1>` tags, and hit return after the closing `<h1>` tag, like this:

```
<header>
  <h1>Artists: home page</h1>
</header>
```

Note the two spaces to indent the `<h1>` tag. Also, make sure the opening and closing `<header>` tags are lined up with each other on the left if the editor hasn't done this for you. Although formatting your HTML code like this is not essential, it is **good practice** to be consistent. It *really* helps you find your way around; you can see a tag's *parents* (the element that contains it), *children* (the tags it immediately contains one level inside) and *descendents* (the tags 'nested' *inside* the *child* tags it contains). We also use two spaces to indent CSS rules inside CSS *rule blocks*.

Now you can style the `<h1>` tag in `styles.css`.

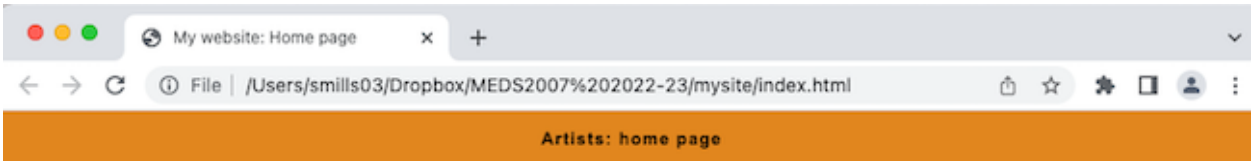
In your CSS file, right at the bottom, on the line after the closing curly brace `}` that ends the `header` styles, hit return and add the following new block of rules. The `header h1` *selector* targets the `h1` tag, which is *nested inside* the `header` tag:

```
header h1 {
  margin: 0;
  font-size: 1.5em;
  letter-spacing: 1px;
  padding-bottom: .25em;
}
```

Targeting a single HTML tag with a single CSS selector will style that tag *everywhere*, but you often want to target a tag *only* when it is nested *inside* another specific tag, which is just what we are doing here with `header h1`. Our style will *only* affect `<h1>` tags that are *inside* a `<header>` tag.

This also means you can use `<h1>` tags outside the `<header>` element (HTML5 allows this in its `<section>` or `<article>` tags, like multiple blog posts or news articles), without this style rule affecting them. You can then style these other `h1` tags separately.

If you save your changes and reload your page you should see that the type size has increased, there is extra space between the letters, and a little more space (in this case `padding`) beneath them.



Picture 3: your page should look like the above image

Before we continue constructing the homepage let's take a look at what all the bits and pieces of the CSS code mean.

Understanding CSS style rules

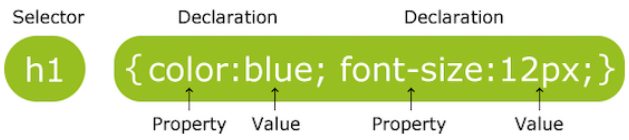
Selectors and declarations

There are two main parts of a CSS rule: a **selector** that targets the correct HTML for the styles (e.g. `header` or `header h1`) defined by a pair of opposite curly braces `{` and `}`, containing one or more **declarations**, each ending in a semicolon `;` e.g. `margin: 0;` and `font-size: 1.5em;`). These two CSS **declarations** mean:

- set the default `h1` margin to zero (otherwise the browser adds a margin)
- set the font-size to 1.5 times the standard font size (which is 1em but larger for `h1`, `h2`... tags)

Properties and values

Each separate CSS *declaration* (we have four in our `header h1` style block) has a *property* (what to style e.g. `font-size:`) and a *value* (how to style it e.g. `1.5em;`). **Important!** the *property* and its *value* are separated by a colon `:` (a space after the colon is good practice) and end in a semicolon `;`. This example from w3schools.com[11] for the `h1` selector for an HTML `<h1>` tag explains the various parts of two CSS *declarations* (here shown on one line):



Picture 4: CSS sytnax

[11] http://www.w3schools.com/css/css_syntax.asp

Types of CSS selector and how to choose names

There are three types of CSS *selector*:

tag for styling specific HTML tags (`h1` in the picture above). A style applied to an HTML tag will style **all** instances of that tag throughout the website.

class for styling *more than one element* on any HTML page. You can use a class as many times as you want. Classes begin with a dot (full stop) instead of a hash symbol (e.g. `.wrapper`).

ID it is good practice to reserve IDs for JavaScript (code for user interaction, not covered here) to target specific HTML elements, but *not* for styling. They only apply to *one element per page*. ID selectors begin with a hash symbol (e.g. `#myidname`).

You can't make up new HTML tag names (except in special cases if you really know what you're doing) but you *can* choose your own `class` (and `ID`) names, as long as they begin with a letter (not a number) and contain only `a-z`, hyphens `-` or underscores `_` (as with file and image names in web design, using only lower-case letters keeps things simple, especially when learning).

Adding more style

Your website doesn't look like much yet, but as you add more HTML elements to the page and apply styles to them, things will begin to take shape.

HTML tag *family* relationships

As already mentioned, it is common to describe contained HTML elements as 'child' (immediately inside a tag) or 'descendent' (inside another tag that's inside a tag, etc.). Elements are described as 'nested' inside the enclosing 'parent' element, and they may 'inherit' the styles of their 'parent' elements (e.g. the `font-family` declaration on the `body` tag will be inherited by all the other elements inside the `<body>` tag) unless a further more specific `font-family` CSS rule (say for the `h1` tag) 'overrides' it. The 'family' relationships all sound very cosy, but it's merely a technical description. Some features of the 'cascade' part of CSS (Cascading Style Sheets) take advantage of this principle of the family-like *inheritance* in HTML.

Let's take a closer look at the style declarations you just added:

`margin: 0;`
: the `<h1>` tag, like many other tags, has default margins. These are good for users when reading text-heavy content, but annoying for designers when an `h1` default margin pushes things out of place in a `header` tag, so it's obvious what this does!

`font-size: 1.5em;`
: makes the type 150% normal size (`1em` is the default, so `1.5em;` is 150%, `.8em` is 80%, etc.). `1em` is the height of one line of text, so using `em` values for `font-size` and `line height` (as you'll see later) as well as for top and bottom `padding` and `margin` values can make it easier to line things up vertically. Professional web designers use this approach to improve the overall alignment of page elements and content.

`letter-spacing: 1px;`
: adds 1 pixel of space between each letter, to space out the text a little and improve readability, especially in headings.

`padding-bottom: .25em;`
: This adds .25 em of space as *padding* to the bottom of the `<h1>` tag, otherwise it sits a little too low when vertically centred with `justify-content: center;` of the header tag. Fine visual adustments like this are crucial in web design. We could have added this with `margin-bottom` too; the only difference is that any `background-color` property is also shown in an element's `padding`, but won't appear in its `margin`.

If you want to experiment with CSS, try other values in the `header h1` CSS block for the `letter-spacing` (e.g. `6px`) or `font-size` (e.g. `1em`). In the `header` CSS you could try another `background-colour`. You can choose some popular colours on this website[12] or use the HTML *colour names*[13].

Be sure to reset any CSS changes back to what they were before your experiments, or the rest of the tutorial might not work as it should!

[12] http://www.color-hex.com/
[13] https://www.w3schools.com/colors/colors_hex.asp

Adding navigation and a content section

Every website needs a menu for navigation, so now you're going to add a `<nav>` tag to `index.html` to contain menu items. It needs to go *after* the `header` tag.

Click just *after* the closing `<header>` tag—the vertical line `|` below shows where to click to place your cursor—and hit return to make a new line:

```
<header>
  <h1>Artists: home page</h1>
</header>|
```

Now type an opening `<nav>` and closing `</nav>` tag and add some temporary text (note: the opening and closing tags need to line up on the left, although the text editor may have done this for you):

```
<header>
  <h1>Artists: home page</h1>
</header>
<nav>
  Navigation here
</nav>
```

In `styles.css` add the `width` and `background-color` for the `<nav>` tag:

```
nav {
  width: 100%;
  background-color: var(--color3);
}
```

Reload the page in a web browser—the boxes are beginning to take shape. The new `<nav>` tag (with the grey background –color3) should stretch across the full 100% width of the page.

Picture 1: your page should look like the above image

The ‘main content’ box

You can now add an HTML `<main>` element for—as you’d expect—the *main content* of the page. A single `<main>` element per HTML page is meant to contain everything unique to that page.

Some content, such as navigation links, contact and copyright information, site logos, site search forms, etc. is typically repeated throughout a website. This kind of content should be outside the `<main>` tag and inside other tags, such as a `<header>` or `<footer>` element.

In the design you are building here, the `<main>` tag goes after the `<nav>` tag.

Click to place your cursor *just at the end* of the closing `</nav>` tag, hit return, then add the `<main>` element’s opening tag, hit return again and type some temporary text (e.g. ‘content here’ so you can see what’s what in the browser), hit return again, then add the closing `</main>` tag, like this:

```
<body>
  <header>
    <h1>Artists: home page</h1>
  </header>
  <nav>
    Navigation here
  </nav>
  <main>
    Content here
  </main>
</body>
```

When all the ‘nested’ code is properly indented with two spaces at the start of the lines, it shows which *descendents* are contained in which *parent* elements so you can see at a glance how the `<header>`, `<nav>` and `<main>` tags are all *inside* the body (`<body>`). This is why it is important to indent your code neatly with spaces.

In `styles.css` you now need a new style declaration with a CSS selector for the `<main>` tag. On a new line below the last closing curly brace `}` of the existing styles, add the following:

```
main {
  /* sets padding: top, right+left, bottom: */
  padding: 1.75em 30px 2em;
  background-color: var(--color4);
  line-height: 1.85em;
}
```

The line `/* sets padding: top, right+left, bottom: */` is a CSS *comment* used to add extra information. Here, it explains one of the padding *shorthand* values, where you set several values in one go. This also works with some other CSS values. CSS comments, enclosed between `/*` and `*/` are ignored by the browser.

By default, the `<main>` tag will appear in the browser underneath the `<nav>` tag because, until they're styled, these 'block-level' HTML elements force line breaks, so appear one after the other in the *same sequence* as they are in the actual HTML code:

Picture 2: your page should look like the above image

This stacking of elements is fine for how we want our design to look on a mobile device, however when viewed on a desktop this design will require the `<nav>` and `<main>` tags to sit side-by-side, left and right, both beneath the header.

Next, we will be creating the code that allows us to style *some* of our elements to appear differently on larger and smaller screen sizes, depending on the kind of device they are viewed on. The code we will use effectively targets the *width* of the screen.

Creating page ‘break points’

The standard way to make a website responsive to the context in which it is displayed, is through the use of CSS *media queries*.

We are going to add a *media query* to our stylesheet to contain styles that will re-style our mobile-sized elements only when the browser width is greater than 600 pixels. That is to say, these styles will be applied when the website is viewed on any device **larger than a mobile phone**.

In your stylesheet, underneath all of the code you’ve added so far, put in the following two lines.

```
/* STYLES "BREAKPOINT" FOR WIDTHS ABOVE 600 pixels: */
@media all and (min-width: 600px) {

} /* <- closing curly bracket of the breakpoint */
```

You can more CSS *comments* between `/*` and `*/`. Comments are really useful to annotate your code to explain what the lines of code below are doing. In this case we are explaining in the comment that the code in this breakpoint will only work in devices larger than a mobile screen.

The second line beginning `@media` is the media query. It basically says that this query applies to `all` types of media devices `and` will work only in browser windows that are *above a minimum width* (`min-width`) of *600 pixels* (`600px`).

Importantly, any styles we want to apply using this media query for larger widths needs to be put between the *two curly brackets* { and } that follow it.

Before we move on to this we need to add one more element to the HTML page. This is a new *parent* box that will contain the `<nav>` and `<main>` boxes as *child* elements. This offers us a way to organise content within a *parent* wrapper, using the CSS *flexbox* model, which is a good way to arrange elements on a page. We will be using it to make some of our elements respond to screen sizes.

Take care at this point, as we will be adding an element *around* the ones we already have. First, below the closing `</header>` tag, but above the `<nav>` tag, add the following opening `div` tag with a `class` attribute, as shown:

```
<div class="wrapper">
```

Then, below the closing `</main>` tag, add the closing `</div>` tag so the whole block of code inside the `<body>` tag looks like the following (note how the extra indentation for `<nav>` and `<main>` shows that they’re now *child* elements of our new ‘wrapper’ `<div>`):

```
<header>
  <h1>Artists: home page</h1>
</header>
<div class="wrapper">
  <nav>
    Navigation here
  </nav>
  <main>
    Content here
  </main>
</div>
```

The div tag needs a `class` 'attribute' so the CSS code can *target* it with the styles you will create for the `.wrapper` class. You will see in a moment how to target a *class name* with a CSS selector.

Because HTML5 does not have a `<wrapper>` tag, the `class` attribute enables a CSS style to 'target' any `<div>` tags with the *same class attribute* (although we will only have one) with a `class` attribute of `wrapper`, but will not affect any other `div` tags.

Using flex to create a responsive wrapper

In this step we will create styles for the `wrapper` class which will change the layout depending on whether the site is viewed on a mobile device or something larger.

First, add the following code to your stylesheet in the section *above* (*not* inside) the `@media` query we just added:

```
.wrapper {
  display: flex;
  flex-direction: column;
}
```

Note that the *selector* `.wrapper` has a 'dot' or full-point in front of it. This indicates that it is the selector for an HTML element that is a member of the class `wrapper`, as is the case with our `<div class="wrapper">`.

The two declarations tell the browser to use CSS *flexbox* for this box, then to arrange any elements *within this box* in a `column`, one above the other. The reason we want these elements in a `column` is that this code is for the mobile version of our site, so we want the elements in the wrapper stacked on top of each other, not side-by-side.

Next, we will add a declaration for the `wrapper` div inside those two curly brackets of our media query, so we can style the wrapper for devices larger than a mobile. Your code block should then look like this:

```
@media all and (min-width: 600px) {
  .wrapper {
      flex-direction: row;
  }
}
```

As you can see above, we have changed the flex-direction from *column* to *row*, so that the `<nav>` and `<main>` tags will appear side-by-side at screen widths above 600 pixels wide.

Save these changes and refresh your page in the browser. To see how the break point works, use your mouse to grab the edge of the browser window so you can narrow or widen the page. You should see when your browser reaches the 600 pixel *break point* as the layout of the page will change.

Below 600 pixels the page will be organised in a `column` to suit mobile devices and will look like this:

Picture 3: your mobile-size page should look like the above image

Above 600 pixels the `<nav>` and `<main>` should be organised side-by-side as a `row` and look like this:

Picture 4: your desktop-size page should look like the above image

Making the main and nav boxes the correct size

Now that we have the boxes realigning to suit their viewing context, it's time to style things to make them the correct size for the wider desktop view. Given that the page can be viewed at different sizes, we need to use a measurement for the `main` and `nav` boxes that changes with the browser window size. The best way to do this is to make the measurements a *percentage* of the width of the browser window.

To do this, we will put the following code for the desktop version *inside* the `@media` breakpoint's curly brackets, immediately beneath the `.wrapper` CSS block we added earlier, to change the width

of the `main` and `nav` tags at larger screen sizes. This will set the `main` box to always be `75%` of the width of the browser window and the `nav` box to be `25%` (note the extra indentation):

```
  main {
    width: 75%;
  }
  nav {
    width: 25%;
  }
```

Your breakpoint should now appear as follows—note how the extra indentation shows how all these *break point* CSS rules are *contained within* the `@media` query:

```
@media all and (min-width: 600px) {
  .wrapper {
      flex-direction: row;
  }
  main {
    width: 75%;
  }
  nav {
    width: 25%;
  }
}
```

Save your changes and reload the page in your browser to play about with the browser width. You will see the structure change in response to the browser window width. Neat!