Front end fundamentals with JavaScript

Couch to Coder | BRIGHT NETWORK

Learning Objectives

- Understand the concept of the client side
- Understand what HTML and CSS is, and how they relate
- Understand how and why JavaScript came to the front end
- Be able to create an HTML page enhanced with CSS and JS
- Be able to request data from a back end server

Intro

Well, this is it - the last piece of our puzzle! I'm not saying we finish everything today - just that as far as full stack development covered on this course goes, this will give you the last missing piece. We will spend today and next week putting everything together, to finally see the big picture!

Our task today is relatively straightforward - we need to build the client side/front end of our application. Straightforward doesn't necessarily mean easy, however. We will do the same thing we did last week - break it down to digestable pieces! Let's start with the concept of the front end.

What is the front end?

As mentioned last time, the front end is probably the easier to understand as far as concepts go. We all interact with apps and websites every day, and the part we interact with is the so called front end. You can draw parallels between working client facing roles and back office when it comes to jobs. The front end's job is to present data, and to be used as an interface for the users of your app.

> Note: When we are speaking of apps, we could mean multiple things - A mobile app, a desktop app or, in our case, a web app. It just means an application, even if in everyday language apps are reserved nowadays for mobile apps!

This means that the front end needs to be able to be accessed by as many users as possible if we want to sell our product. While in the 90s and before many apps were distributed on different disks (think the floppy disk, CDs, DVDs, etc.), since the early days of the internet, the web browser became the standard. The main reason of course is the wide range of websites that could be accessed using the browser.

In order for this to be a uniform platform for companies to distribute their services and for developers to create apps on, we need uniform tools as well - enter HTML, CSS and JavaScript. Let's start with how they relate to each other!

The house analogy

One of the easier ways to visualise how the different parts interact with each other is to imagine them as the elements of a house.

HTML - the structure

It defines the walls, floors, doors, and windows - essentially the basic layout and content of the house. In web terms, HTML provides the structure of a webpage, such as headings, paragraphs, images, forms and buttons. HTML stands for HyperText Markup Language. It is not a programming language. It basically annotates raw text that can later be displayed and interacted with using CSS and JavaScript. Almost all websites use HTML as their core structure.

CSS - paint, furniture, decorations

It makes the house look good by adding colors, styles, and layouts. In web terms, CSS is responsible for the visual presentation, like fonts, colors, spacing, and overall design. CSS stands for Cascading Style Sheet, the reason for that is that high level CSS styling applies to elements nested inside the targeted elements as well - cascading down like a waterfall. It is also not a programming language, although most recent developments include certain scripting abilities to target text more efficiently.

JavaScript - plumbing, electricity, smart features

While you can live in a house without the above, it would act more like a temporary shelter rather than an actual house. All the convenient and modern features that houses can provide are behaviours that make good houses great - same with JavaScript.

We will build up our front end in this order as well.

HTML

While technically we can create a full website in a single file, nesting all needed HTML, CSS and even JS in it, it is heavily discouraged in modern web development - it's more common to break it down futher into multiple files. For our simple app, we will keep short and straightforward, with one file each.

Let's create our `session_4` folder, and create an `index.html` file in it. Inside the file, we could create HTML code from scratch, but to keep things simple, I recommend typing in `html5` then hitting enter/tab, to autocomplete a boilerplate file - this will become exponentially more important to be used with HTML, otherwise we might miss a closing tag! Another important thing is to remember to use indentation - with each new tag going on a new line, our code could easily become unreadable!

The important bits are the following:

1. Everything inside the `head` tag is invisible on a website - the only difference is the `title`, which will appear on a tab, and search engines also use this as the clickable link when listing the page.
2. Everything inside the `body` tag will be displayed, which we can break down further.

3. Leave the `meta` tag - without it, we cannot make responsive websites!

There is plenty of things to learn about `html` - we will keep it short to focus on the bigger picture. Let's cover some basics!

Most websites follow the classic breakdown of creating 3 distinct parts - the header, footer and main areas. Inspect a well known website of your choosing - apart from some important stylistic differences, this is pretty common accross all pages! We could follow this like so:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <header>

  </header>
  <main>

  </main>
  <footer>

  </footer>
</body>
</html>
```

It's important that as far as functionality goes, the used tags themselves don't "matter" - you could use only `div` or `header` tags throughout, and you could create a functional website - however, this can mess with multiple things, including SEO (Search Engine Optimiziation), screen readers, and most importantly, the maintainability of your website - if you are not using the appropriate, semantic tags, things will quickly fall apart!

We will slowly build up functionality from the top down. We will start with a header, then create the main area of our app, then finish with the footer.

The header will include an `h1` tag that will act as our logo, a couple of links, and a `Recipes` button that will load in our recipes in the future!

```html
  <header>
    <h1>InstantGramEn</h1>
    <a>Recipes</a>
    <a>About us</a>
    <a>Contact</a>
  </header>
```

Creating an anchor tag makes the text between the tags a clickable link. We will not add the target the target url, which would be the `href=` tag - we might later!

The next step is adding the main area - we will keep it simple, because we will use JavaScript to populate most of our code here!

```
<main>
  <h2>My Favourite Recipes</h2>
  <ul>
  </ul>
</main>
```

Generally it's good practice to only have a single `h1` tag per website, as it acts like a title - then the rest is more like chapter headers, so we can use more! We created an empty `ul` tag - `ul` stands for unordered list, this is conventional for items which will be listed. In our case, we will have a list of recipes displayed here.

> Note: Some tags will display something even with no value, while others, especially ones with opening and closing tags, it is necessary to add something in the middle.

We will keep the footer simple:

```
<footer>
  <a>About us</a>
  <a>Contact</a>
  <a>Terms of use</a>
</footer>
```

Usually the footer is used for navigation and additional info about the site.

Everything looks a bit wonky now, but do not fret - we will address this soon! Structuring our HTML is the job of our CSS file.

Let's open up our file - doing it the old fashioned way however has some downsides. If we want to keep the browser window up to date, we would have to constantly keep refreshing the browser. To circumvent this problem, let's install another extension called `Live Reload`!

You will see the launch button in the bottom right corner - press it, it should open your browser window. Now after each save, the webpage reloads automatically!

CSS

Next step is to add our CSS - this will require 2 steps before we can proceed with anything.

First, we need to create a CSS file. Let's create it right next to our html. We could put it anywhere we want in our project, as we have the ability to link from a path relative to our HTML file, but this will be good enough for us. Let's call it `style.css` - you will soon see why!

Next, we will establish a connection between the 2 files - a good trick is to change the background colour in the file to something bright and bold, this way, when we finally connect the two, we will see the change immediately!

In our file, add the following:

```css
body {
  background-color: hotpink;
}
```

Now, we will tell our HTML file where is the CSS file located. Add the following to our `head` tag:

```html
  <link rel="stylesheet" href="style.css">
```

This is where we would add a relative path if needed, but as long as the files live next to each other, we do not need to change anything!

If everything went well, after saving both our files, we should see the background colour change to a very bright shade of pink!

So what is happening exactly?

Whenever you are writing in a CSS file, what you are really writing is a so called `selector`. These selectors can get quite complex, depending on how specific is your CSS query, but it can be as simple as just writing a single HTML tag.

If something is nested inside another tag, you can depict that with the `body > h1` syntax, or the `body h1` syntax. The difference is that the first only refers to h1 tags directly nested inside the body tag, while the other targets every h1 tag inside the body, no matter how many layers are between the two. This can have some heavy consequences - you might change the style of some element that you did not intend to!

Let's modify a couple of things, like colour, font, and margins!

```css
body {
  font-family: "Open Sans", sans-serif;
  margin: 0;
  background-color: #F8F8F8;
  color: #333333;
}
```

This gives us nicer contrasts and a good, easy to read font.

Margins, paddings and borders are all part of the so called `box model`. We will modify these for a good look, but if you are interested in how they work, they definitely are worth reading up on on the MDN docs: MDN - Box model

To make things look better, it's worth positioning both your headers and your footers and make them fixed on the page - this is where you can make good use of your chosen AI chatbot - short, precise questions tend to get better answers, but be prepared that the almighty GPT is certainly capable of being wrong!

Add the following for the header:

```
header {
  background-color: #6A9C89;
  color: white;
  padding: 10px 20px;
  text-align: center;
  position: sticky;
  top: 0;
  z-index: 1000; /* Ensure the header stays above other content by putting it
forward in the 3rd dimension*/
}
```

And the following to the footer section:

```
footer {
  background-color: #6A9C89;
  color: white;
  text-align: center;
  padding: 10px;
  margin-top: auto; /* Push the footer to the bottom */
}
```

Without the actual recipes, this will still look slightly wonky, but we are slowly, but surely, getting there!

Next, we will need to start thinking a bit about creating the logic for our app. This is where JavaScript will come in!

But why do we use JavaScript in the browser? Let's have a little history lesson!

What do we know about the circumstances of the birth of JavaScript? Let's see!

- **Created in 1995** by **Brendan Eich** while he was working at **Netscape Communications**.
- Originally named **Mocha**, it was later renamed **LiveScript** and finally **JavaScript** to capitalize on the popularity of Java at the time (apart from some light similarities in conventions, they have nothing to do with each other!)
- JavaScript was designed to be a **lightweight scripting language** for the web, utilising the client's computing power for interactivity.

Why was there a need for the language?

- **To Make the Web Interactive**:
    - In the early days of the web, pages were static and lacked interactivity. JavaScript was created to allow developers to add dynamic behavior to web pages, such as form validation, animations,

    and user interactions.
- **To Run in the Browser**:
    - JavaScript was designed to run directly in the browser, eliminating the need for server-side processing for simple tasks. This made web pages faster and more responsive.
- **To Complement HTML and CSS**:
    - While **HTML** provided structure and **CSS** added style, JavaScript brought functionality, completing the trio of technologies for building modern websites.

The fact that the language comes baked in with every single browser means that developers can almost guarantee that JavaScript will operate on their client's browsers, thus they can utilise its power!

Let's add our JS file! We need to create, similarly to our CSS, a file, and then we need to link it to our HTML. We can call our JS file `app.js` as per convention, but of course, there are multiple good options here. Linking it is done using the `<script>` tags in the `head` tag, we can safely put it next to the CSS link tag.

```
  <link rel="stylesheet" href="style.css">
  <script type="module" src="app.js" defer></script>
```

But how can we be sure that our code is running?

Let's add a little console.log and let's check the inspect option in our Chrome browser - voila, our code is up and running!

The reason for the `defer` keyword is touched on in the video a bit more in depth - in short, we need to make sure our HTML is loaded before our JavaScript, otherwise we would have no knowledge of what HTML elements to modify, since it loads in so fast!

Our last task is to use JavaScript to modify our code. For now, we will just populate it with some dummy data, but come next week, we will attach ourselves to our own server-side code!

Before we get started, we will try to visualise how our recipes might look like. Essentially, every recipe must be uniform, since their data will (ideally) be uniform. This means we will design a small HTML "card" that we will display (think of it like an article, a social media post or a product on an online webstore - these are identical, repeatable elements that share a similar shape with title, details, description, price, etc.)

For us, it will look a little something like this:

```
<div>
  <h3>Title<h3>
  <p>Time</p>
  <p>Cuisine</p>
  <ul>
    <li>Ingredient 1</li>
    <li>Ingredient 2</li>
  </ul>
  <ul>
    <li>Step 1</li>
    <li>Step 2</li>
  </ul>
</div>
```

But this will not be copied into our file - the reason for that is that we would like to keep the number of recipes dynamic. We don't know how many recipes there will be, or which one will be asked to get loaded into the browser. For that reason, we will use JavaScript to generate these elements.

Let's add the following code to our `app.js`, and then we will discuss what these lines are doing!

```
const recipeList = document.querySelector("ul");

const titleTag = document.createElement("h1");
titleTag.textContent = "Spaghetti Carbonara";
const timeTag = document.createElement("p");
timeTag.textContent = "Time: 30";
const cuisineTag = document.createElement("p");
cuisineTag.textContent = "Cuisine: Italian"
const ingredientListTag = document.createElement("ul");
const ingredientListElementTag = document.createElement("li");
ingredientListElementTag.textContent = "Get all ingredients";
ingredientListTag.appendChild(ingredientListElementTag);

const stepListTag = document.createElement("ul");
const stepListElementTag = document.createElement("li");
stepListElementTag.textContent = "Do all the steps";
stepListTag.appendChild(stepListElementTag);

const recipe = document.createElement("div");
recipe.appendChild(titleTag)
recipe.appendChild(timeTag)
recipe.appendChild(cuisineTag)
recipe.appendChild(ingredientListTag)
recipe.appendChild(stepListTag)


recipeList.appendChild(recipe)
```

This will look a bit repetitive, but bear with me - with enough time, there are techniques to be learned that removes some of the repetition!

First of all, we need to break down a recipe to its building blocks: each one will have a title tag, time tag, cuisine tag, list of ingredients, and a list of steps.

If we create an HTML tag from scratch, we have to use the `document.createElement()` method - by passing in the name of the element we want to create, JavaScript will create an empty element for us in the background, be it a header tag, or a list element.

Then we need to give content to the elements that need it - things like the `p` tags or the `li` tags need text content.

Finally, we need to assemble it in the following fashion (from the top down when looking at the code):

1. First, we grab the manually created `ul` element that we will use to add the recipes to.
2. After that we create a `div` element - this will act as the container for our recipe.
3. Then we create the title, time and cuisine tags, and also set their innerText properties - giving them values.
4. Next, we create the 2 nested unordered lists
5. We create list elements for ingredients and steps, respectively
6. Finally, for the final assembly:
    1. We first add the list elements with `appendChild` to the unordered lists
    2. Then we add the elements in order to the `div`: Title -> Time -> Cuisine -> ingredient list -> step list
    3. Finally, the recipe itself need to be added to the recipelist at the top with the same appendChild!

If everything aligned perfectly, we should see a dumbed down version of our carbonara recipe!

Even though it is clear that this is hard to maintain and repetitive, but still, it gives us a nice, JS-generated block of html to display, and it also gives us a chance in the future to load in all the recipes and, using a for loop, display them, without us requiring to increase the lines of code!

On the next week, we will do a multitude of things:

- We will learn how to load in recipes from our back end
- We will add a form to our front end so we can save new recipes
- finally we will change the back end so we can store the freshly created recipes.

See you next week!