YOU CAN MAKE A BAD WEBSITE!

LESSON TWO: FLEXBOXES AND MORE STYLING

A Shitty Guide by Zenith

(It looks scarier than it is.)

DISCLAIMER

I am a hobbyist webmaster and I don't know JavaScript. So this series of documents will solely deal with HTML and CSS capabilities. This isn't intended to be used as if it's a master guide. As the title implies, I strongly believe in "you can make it better later, but first you have to make it exist." My own coding style is pretty sloppy, and I frequently have to access online guides for help while working. This is just as much a refresher and practice for me as it may be for you.

Everything is difficult until it is easy. When first learning HTML and CSS I was confused all the time and easily discouraged by others having websites that I found prettier than my own. Always remember that the shittiest indie website has more heart in it than the best corporate owned platform. It's okay to ask friends for help, to look at tutorials, and most importantly, to accept you fucked something up beyond repair and start over. The good thing about being alive is that the point of it is to learn. Don't give up, always take breaks, and remember it's only text and there is no grade or money on the line.

TABLE OF CONTENTS

LIVE PREVIEW AND NOTATED CODE

This tutorial will be linked alongside a live preview showing what will be created using the instructions of this tutorial. There will be an additional link to a .zip file containing the HTML, CSS, and other files that went into creation of these pages with notation added. You will note that within the .zip file there will be HTML documents and CSS styling for elements not covered in this document. I have premade these files to make writing future tutorials less stressful. If you feel like you're getting a good grasp of the structure of HTML and CSS from these two tutorials alone, by all means, work ahead!

HOW FLEXBOXES WORK

As promised in lesson one, I will be building a simple two column display website, or to think of it in another way, a website with a body and a sidebar. First, let's return to my explanation of how flexboxes work from the first lesson.

> "Flexboxes are created by styling a with the declaration “display: flex;”. A flexbox is a way to lay out your s and other elements. It is one-dimensional, meaning it only controls the layout of a column or a row, but not both at the same time. The direction a flexbox modifies can be specified using further CSS declarations. When you style a to have the flex property, the s placed inside it will place themselves in a line. It is also possible to cause flexboxes to wrap elements inside them when they are too large to display on a single line."

All of this information is still true, but let's go further. In the flexbox display we'll be making today, we will **not** be altering the flex-direction property, meaning it will display using flexbox's default layout: rows. Why are am I referring to this project as a two column display if it's constructed from rows? **When visually looking at the final product, it looks like two columns standing side by side because of our styling.** The browser is (correctly) reading the code to put two elements to the left and right of each other. Conversely, when the flex-direction is set to "columns", it reads it as placing elements in a row vertically.

When we look at this screenshot of the live preview of what the site I will be making looks like, we can see that it looks like two columns standing side by side.

In actuality, this is technically a row of two div elements. But doesn't it make more sense to describe it as a two column lay out? Let's go over how to set up this basic page structure.

SETTING UP A TWO COLUMN FLEX LAYOUT

In order for this to work correctly, we have to make sure that we're styling our HTML elements correctly. Personally, I start with the CSS of flexbox layouts before writing the HTML and tweaking it once I can see the visuals of what I'm working with. This is ultimately preference based and starting with one over the other is not technically wrong.

For this section of the tutorial, we will be ignoring purely aesthetic elements such as the border-image and fonts used on the live preview. They will be covered in a later section of this document to avoid being overwhelming. On the live preview of this site, I used a long string of "lorem ipsum" placeholder text for the body to show how large chunks of text look under these instructions. **I will not be using that in this tutorial**. I will simply be writing one sentence of place holder text to avoid ripping my hair out.

PART ONE: THE HTML

First, we will be creating a div container for our flexboxes to appear in. In this example, I will be naming the div **flexarea**. We would write it like this:

```
<div class="flexarea"> </div>
```

On its own, this class will only be defining the elements within it as being displayed in a flex layout within the area it designates in its **width** and **height** declarations.

Next, we need to create the areas of our website that will be displayed in our **flexarea**. I will be calling them **section1** and **section2**. These sections will go **inside** the **flexarea** div. If they are placed outside of it, then they will not display in a flex layout.

```
<div class="section1"> </div>

<div class="section2"> </div>
```

These two elements will display from left to right. So, **section1** will be on the left of **section2**. Let's put some text inside each element so we can see what's in them.

```
<div class="section1><p>This is a text line.</p></div>

<div class="section2"><p>This is a text line.</p></div>
```

Now, our HTML document should look like this:

```
<div class="flexarea">

<div class="section1><p>This is a text line.</p></div>

<div class="section2"><p>This is a text line.</p></div>

</div>
```

If you don't want a header or a footer on your page or any decorative elements addressed later, technically, this is the bare minimum of the HTML you would need to write for it!

PART TWO: THE CSS

Before we do anything else, we need to style the **flexarea** we created so the other elements within it display in a flex layout. The most important part of this CSS will be that it is set to **display: flex**.

```
.flexarea {
width: 100hw;
height: 100hv;
display: flex;
margin: auto;
overflow: scroll;
}
```

Notice how the **width** and **height** are set to 100hw and 100hv respectively. That means the sections we put within that div are going to naturally take up 100% of the browser's visible screen. Since we've specified the width and height we want, when we style the inner sections, we can use percentages of that size in the CSS to make it slightly more responsive on mobile.

Now that we've styled the flexbox itself, we can start styling **section1** and **section2**. I didn't want my two sections to be evenly sized because I used the second one as a sidebar. I made I also wanted **section1**, the one on the left, to be the larger one.

```
.section1 {
width: 85%;
padding: 1%;
margin: 1%;
}
```

We've designated 85% of our screen size to **section1**, so to keep things from clipping into each other, we need to make sure **section2** is no wider than 15%.

```
.section2 {
width: 14%;
padding: 2%;
margin: 1%;
}
```

I used a width of 14% to make sure everything was going to work with each other nicely. Having finished this CSS, you have now technically created a flexbox layout. Now, let's add a header and footer to work as a title and place to store additional information (like credits) before we work on styling things to look prettier.

MAKING A HEADER AND FOOTER

In this layout, our header and footer elements need to be outside of the **flexarea** in order to display correctly. These are not required for the flexbox layout to function, but serve as helpful additional elements to make your website easier to navigate (and look better.) In this example, the CSS for the two elements will actually be the same for both.

I had to zoom out really far to get both the header and footer in frame, the text on this website is not naturally this small.

PART ONE: THE HTML

The header element will be placed before the **flexarea** div begins. Conversely, after closing the **flexarea** div, you will place the footer element after it. Most of the magic in our header and footer elements will come from the CSS styling. However, as a refresher, I will go over how to write their tags and give a rough look at their placement in your HTML page.

```
<header><h2>Title of Page</h2></header>

<div class="flexarea"> ... </div>

<footer><p>This is the footer with credits and more.</p></footer>
```

The **flexarea** div and everything inside of it will lie between the header and footer elements. In terms of HTML, it's really that simple!

PART TWO: THE CSS

Since our header and footer will be using the same styling rules, we don't need to create two separate CSS selectors for them. We will simply use the same declarations for both sections by separating the names of the two element selectors with a comma.

```
header, footer {
margin: 1%;
height: fit-content;
text-align: center;
}
```

See the **fit-content** declaration? That means the header and footer's heights will expand to match the height of the elements we put inside it. However, once it reaches the maximum height of it's parent element, it remains within those boundaries. Since we don't have a parent element for our header and footer, it uses the HTML document itself as a parent, meaning that it can get as tall as it needs to to contain the elements in it. The only limitation to the height of these elements are that they both have a margin of 1%, meaning that they will always leave that 1% margin around their height to avoid clipping into or sitting too closely to other elements.

DIVING DEEPER INTO STYLING

So, as promised earlier, I will show you how to turn this layout from plain and ugly into something pretty cool. Since this is mostly CSS, I'll be breaking into sections explaining how each of these declarations will work. The exception to this will be the addition of a decorative image to the sidebar, as it will require minor HTML changes too. Its section will simply include an HTML breakdown of the changes, too.

USING LOCAL FONTS

When I was first learning CSS, trying to use local fonts was exceptionally confusing for me. Hopefully, I will be able to explain it and give advice that will prevent you from spending two hours googling things like I had to do. Let's take a look at the CSS for loading our fonts into our website.

```
@font-face {
font-family: Alte;
src: url(/AlteHaasGroteskRegular.ttf)
   }
@font-face {
font-family: Flowers;
src: url(/FlowersKingdom.ttf);
}
```

The live preview I am creating today uses two fonts: one for the paragraph text, and the other for heading text. There is not technically a limit to the amount of fonts you can load into one site, but I would suggest limiting yourself to three or four at most.

The most important part of this CSS is making sure that we are linking to our fonts using the correct file paths. In this example website, I have not used any folders and therefore each font can be directly linked to like this. It is important to remember to place copies of the fonts you are using into your main website folder because that is what the code will be reading once you upload your site to a webhosting service. It can't read the files that are on your personal computer anymore, and has to look for them in the files that it was uploaded with. Make sure to remove any spaces in the file names of your font files. I am not sure if this is a universal issue, but when trying to load fonts with a space in their file name, it never works for me. So this is easily fixed by simply renaming files to avoid spaces.

Once we've linked the source of our font, we then need to declare what its font-family will be called. This name will be what we use to style our paragraph and heading types as the fonts we want them to be. I shortened the name of the font file names to work as my font-family titles because it makes it easy to quickly identify which file is being called if I were to change it in the future.

STYLING THE BODY AND HEADING TEXT

By styling our body element, we're giving the website a background and changing the default color of text. We will then go on to change the color of heading text to make it different from paragraph text to allow it to stand out better. On this example site, I have only used a color for the background, however, by using background-image, you can create a tiled background using an image or gif.

```
body {
background-color: #70c1b3;
color: #247ba0;
font-family: Alte;
}
```

The **color** declaration refers to text color, rather than the color of the background. As you can see, we're using the font-family named we named "Alte" as our standard/paragraph font.

For our heading text, we will be changing the color and font to the shade of pink you see on the header and as the title of **section1**. I will be making these declarations to two sizes of heading text. Since we're not changing their font size, we can use the same trick as we did for our header and footer elements.

```
h1, h2 {
color: #f25f5c;
font-family: Flowers;
}
```

Now, the heading text will be pink and a different font than our paragraph text, making it easy to skim pages and see what you're going to be reading about.

THE BORDER-IMAGE PROPERTY MADE SIMPLE

Learning how to use the border-image property was a nightmare for me as every single online tutorial failed to explain how the size of your border directly impacts the way the image you're using as the border displays on screen. This section will likely be more dense than the other styling skills you will be reading about as this property appears very complex but is actually very simple.

This website uses border-image to create the cute scalloped edge you see around the header, footer, and the two sections in the flexbox. Let's look at the CSS it took to do it, and then go over how the border-image property is functioning. The CSS below is the CSS used on each of those elements to create the effect, there are no changes between each element's border image. You can add a border-image to any of the previously created divs by understanding the CSS below.

```
border: 45px solid;

border-image: url(border.png) 42% round;

background-color: #ffe066;

background-clip: padding-box;
```

Now, let's go over what each line is doing.

First, we need to declare a border on an element. Since we're working with border-image, we need it to be thick in order for the effect of the image to be visible. 45px tends to work nicely for most images. But you can alter this number to be larger or smaller depending on the look you want.

Border-image is linking to an image file. Let's look at the image it's linking to on this site and explain what's happening to it.

This is what our border-image looks like.

So, how is this circular image being turned in the scalloped effect of our rectangular divs? Our image is being sliced into nine pieces and each of those pieces are being stretched, duplicated, or otherwise manipulated by our rules to to replace the thick border we created earlier with an image matching its size. Four of these pieces will form the corner on our rectangles. The absolute middle piece is discarded and we don't see it in our border. However, the edge pieces between each corner are being repeated and/or scaled to create our border's scalloped edge effect. We are specifying percentage of the side of the image

each edge slice should take up. In our case, we are creating four edge pieces that are 42% of the size of the side of the image they are on. Then, by specifying it to be "round", we are repeating those edge slices until all the gaps of the border are filled, resizing our slice to fit the space instead of cutting itself off if it is too large.

Now that we've created a cute scalloped edge, we have to change the background color of the element to prevent it from just being a hard yellow line where the border stops, and the blue of the body element begins. To do this, we're just going to use the hex code for the yellow of the border and make it the background image of our element.

After adding a background color, you'll notice it will now clip into the border image, rendering the element a plain rectangle again. We can fix this by specifying that the background color should stop being filled in once the padding of an element begins. This allows the border's scalloped edge to be visible again!

ADDING AN IMAGE TO YOUR SIDEBAR

This section will require some simple HTML additions as well as some CSS in order to function. While there are many ways to do this, I will be showing you how to achieve it by creating a div for the image, and another div for the text as it is easier to organize and style that way. We are going to be creating two div elements that will be placed within our **section2** div.

PART ONE: THE HTML

```
<div class="section2">

<center><div class="s2img"><img src="Your image link" alt="A description of your image."></div></center>

<div class="s2text"><p>The text you're putting in the sidebar.</p></div>

</div>
```

In the code snippet I will be providing, you will see that there are multiple paragraph text lines. This is so each line could be made into a link if you wanted to link other pages or social media accounts on it. While there are more formal ways to create lists of links, this way personally is the easiest for me to manage and style effectively.

PART TWO: THE CSS

This CSS styles the HTML we just created. We are creating a container for the image with a specified width of 80% so the image we place in it doesn't overflow outside the sidebar. Then, we're styling the image placed inside that div to take up 100% of the allowed space and to align itself with the middle of the div rather than the left side. Finally, we specify that we want the text div we've created to inherit its parent element's width, aka we're having the text element take up the entire width of the sidebar (14% of our total page width.)

```
.s2img {
width: 80%;
vertical-align: middle;
}
.s2img img {
width: 100%;
}
.s2text {
width: inherit;
}
```

STYLING LINKS AND HIGHLIGHTED TEXT COLOR

Finally, let's style the links and what color text turns when you highlight it with your mouse. This CSS is very easy to do and is a fun way to make your website more unique.

```
a {
color: #f25f5c;
}
a:hover {
color: #247ba0;
}
a:visited {
color: #247ba0;
}

::selection {
    color: #247ba0;
    background: #70c1b3;
}
```

As you can see, we're just specifying the tag for links (a) and altering their color when they're in different states. The selection section allows us to make the color of the text change when highlighted, and provide a solid background of another color to make it even more clear that the text is being highlighted.