Edinburgh Napier University

SET08101/SET08401 Web Tech

Lab 5 - Building Page Layouts with CSS

Dr Simon Wells

Aims

At the end of the practical portion of this topic you will be able to ::

- Use the CSS flexbox and grid layouts to influence how the elements that make up your page are presented

NOTICE: Like everything else to do with the web there is a lot of useful material online. In addition to reading the relevant chapters of the module texts, you should also avail yourself of the following which document CSS:

- `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS3`
- `https://www.w3schools.com/cssref/default.asp`
- `https://devdocs.io/css/`

Part I
CSS Flexbox & CSS Grid Layout

HTML is a great embodiment of the Hypertext idea. Treated simply, HTML will reflow to efficiently present text documents and associated metadata in an elegant way, regardless of screen size, window size, client browser, or user device. However, despite this, and despite the fact that HTML works really well in this respect, and isn't designed to be a pixel-perfect layout, designers and developers have consistently tried to get HTML to do things that it was never designed to do. For example, to emulate the pixel perfect layout engines that we are used to when developting desktop applications. This leads to frustration, it is really hard to achieve, and to misusing existing elements to try to get consistent and exacting behaviour.

This leads to increasingly complex solutions, and broken features across different clients, as well as weird edge-case behaviour when windows are bigger or smaller thant the designer considered, or when a client is non-standard, for example, visually impaired users will often have a range of tactics to make the Web more usable for themselves. These tactics can range from personalisd CSS that is applied to every page that they browse to, in order to control contrast of colour, size of text, etc. right through to screen readers or braille output devices. As a result, if you try to finely controll the presentation of your information and go beyond what HTML was designed to do, then you risk introducing accessibility issues.

To enable more exacting layouts, whilst maintaining accessibiity, CSS Flexbox and CSS Grid Layout were introduced.

Activities

CSS Flexbox

Flexbox is a layout model that enables developers to organise itesms in one-dimension and to manage the space between items. This means that flexbox handles layout of elements one dimension at a time, that is, as a row or as a column. Later, you'll want to contrast that with the two dimensional model of the Grid layout which enables handling of rows and columns together.

Let's start with some simple boxes that we want to arrange horizontally so that they flow nicely across the screen and are arranged nicely as the window is resized. First, some HTML; create a file named flex1.html and edit it as follows:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Modern CSS</title>
  <link rel="stylesheet" href="flex1.css">
</head>
<body>
<ul class="flex-container">
  <li class="flex-item">1</li>
  <li class="flex-item">2</li>
  <li class="flex-item">3</li>
  <li class="flex-item">4</li>
  <li class="flex-item">5</li>
  <li class="flex-item">6</li>
</ul>
</body>
</html>
```

Notice we've assigned a flex-container class and a flex-item class to the unordered list and list item elements respectively. These aren't special, just class names to hook our CSS into. Notice that we've also got an external css file.

Now some CSS to style our HTML. We'll store it in flex1.css

```
.flex-container {
  padding: 0;
  margin: 0;
  list-style: none;
  display: flex;
  flex-flow: row wrap;
  justify-content: space-around;
}

.flex-item {
  background: tomato;
  padding: 5px;
  width: 200px;
  height: 150px;
  margin-top: 10px;

  line-height: 150px;
  color: white;
  font-weight: bold;
  font-size: 3em;
  text-align: center;
}
```

Notice that we have two CSS blocks here, one styling the flex-container and the other styling the flex-item. Much of the content is just CSS styling and actually has little to do with the flexbox, just some CSS to make things look pretty and help demonstrate the flexbox. You should experiment with altering and commenting out the items to get a feeling for what effet the various CSS properties have. The only lines that are important with respect to the flexbox are:

```
display:flex;
flex-flow: row wrap;
justify-content: space-around;
```

These do the following *display:flex;* sets the display property for the referenced element to *flex*, then *flex-flow* is used to set both flex-direction and flex-wrap properties. Finally *justify-content* is used with the *space-around* value to get the flex layout to add space before, between, and after elements. There are other values for these properties that you should investigate within the CSS Flexbox documentation.

We can use the flexbox to produce quite pleasing layouts for our user interface. For example (flex2.html):

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>3-columns layout with full-width header and footer</title>
  <link rel="stylesheet" href="flex2.css">
</head>
<body>
<div class="wrapper">
  <header class="header">Header</header>
  <article class="main">
    <p>Pellentesque habitant morbi tristique senectus et netus et malesuada fames
        ac turpis egestas. Vestibulum tortor quam, feugiat vitae, ultricies eget,
        tempor sit amet, ante. Donec eu libero sit amet quam egestas semper. Aenean
         ultricies mi vitae est. Mauris placerat eleifend leo.</p>
  </article>
  <aside class="aside aside-1">Aside 1</aside>
  <aside class="aside aside-2">Aside 2</aside>
  <footer class="footer">Footer</footer>
</div>
</body>
</html>
```

Note that the faux Latin is a technique called *greeking* that is used to provide placeholder text when constructing a user interface mockup before the copy has been written or finalised. Notice how the HTML defines a bunch of containers within the body, all wrapped in a pair of ¡div¿ tags. These containers give us a header, article, two asides, and a footer container. All we are doing is using HTML elements to wrap our content so that it is logically separated into named areas. Each named areas has an assigned class which we can exploit using CSS (& flexbox).

Let's do that using some CSS (flex2.css):

```
.wrapper {
  display: flex;
  flex-flow: row wrap;
  font-weight: bold;
  text-align: center;
}

.wrapper > * {
  padding: 10px;
  flex: 1 100%;
}

.header {
  background: tomato;
}

.footer {
  background: lightgreen;
}

.main {
  text-align: left;
  background: deepskyblue;
}

.aside-1 {
  background: gold;
}

.aside-2 {
  background: hotpink;
}

@media all and (min-width: 600px) {
  .aside { flex: 1 0 0; }
}

@media all and (min-width: 800px) {
  .main    { flex: 3 0px; }
  .aside-1 { order: 1; }
  .main    { order: 2; }
  .aside-2 { order: 3; }
  .footer  { order: 4; }
}

body {
  padding: 2em;
}
```

Of note here, for each element we've set a different colour and used some padding to clarify the different areas for demonstration. The wrapper class sets up our display and flex-flow values. Notice that we've also introduced s the *@media* rule to enable us to apply rules based upon the width of the screen. What is really useful here though is the order property which is used to specify the order in which our containers are displayed.

Now let's take a quick look at how we could use this to construct a menubar type layout (flex3.html):

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Flexbox Navbar</title>
  <link rel="stylesheet" href="flex3.css">
</head>
<body>
<ul class="navigation">
  <li><a href="#">Home</a></li>
  <li><a href="#">About</a></li>
  <li><a href="#">Products</a></li>
  <li><a href="#">Contact</a></li>
</ul>
</body>
</html>
```

That's fairly straightforward, a simple HTML page with an unordered list containing hyperlinks to each page that we want to be able to navigate to. Note that the hyprelinks are just placeholders using # for the moment, but you could replace each with a link to another page.

Now our CSS (flex3.css):

```css
.navigation {
  list-style: none;
  margin: 0;

  background: tomato;

  display: flex;

  flex-flow: row wrap;
  justify-content: flex-end;
}

.navigation a {
  text-decoration: none;
  display: block;
  padding: 1em;
  color: white;
}

.navigation a:hover {
  background: darken(tomato, 2%);
}

@media all and (max-width: 800px) {
  .navigation {
    justify-content: space-around;
  }
}

@media all and (max-width: 600px) {
  .navigation {
    flex-flow: column wrap;
    padding: 0;
  }

  .navigation a {
    text-align: center;
    padding: 10px;
    border-top: 1px solid rgba(255,255,255,0.3);
    border-bottom: 1px solid rgba(0,0,0,0.1);
  }

  .navigation li:last-of-type a {
    border-bottom: none;
  }
}
```

With just a minimum of flexbox to control layout and some other CSS to make things pretty, we are able to get a nice looking nav bar at the top of our screen. This time we've used the justify-content property with the flex-end value to get the nav bar elements to appear to the right of the screen.

CSS Grid Layout

This is another way to use the display property to cause an HTML element, and it's children to be treated as a group which are then laid out accordingly. The group in a grid layout, is, perhaps unsurprisingly, laid out in a Grid or rows and columns. This gives us two dimensions which contrasts with the flexbox layout. Whilst the flexbox is concerned mostly with the ordering or elements and application of space between and around them in a single, linear dimension, grid layout orients the elements in two dimensions, rows and columns, and let's us control the order of the elements and the spaces between and around them.

Let's start with a simple example (grid.html):

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" href="grid1.css">
</head>
<body>

<div class="grid-container">
  <div class="grid-item">1</div>
  <div class="grid-item">2</div>
  <div class="grid-item">3</div>
  <div class="grid-item">4</div>
  <div class="grid-item">5</div>
  <div class="grid-item">6</div>
  <div class="grid-item">7</div>
  <div class="grid-item">8</div>
  <div class="grid-item">9</div>
</div>

</body>
</html>
```

All we have here is a div element which is our container for the things that we want the grid layout to manage. We then have a number of child div elements, each of which will be an item within our grid layout. We've used grid-container and grid-item as class identifiers to help us reference the collcetion and individual items within it.

Now some CSS to style our grid. It's worth seeing what that page above looks like before we have any styles applied so that we can see just how much work is being done by grid layout and our ancillary CSS to turn the HTML into a nicely presented page.

```
.grid-container {
  display: grid;
  grid-template-columns: auto auto auto;
  background-color: tomato;
  padding: 20px;
}
.grid-item {
  background-color: rgba(255, 255, 255, 0.8);
  border: 1px solid rgba(0, 0, 0, 0.4);*
  padding: 10px;
  font-size: 30px;
  text-align: center;
}
```

The .grid-container class has some properties applied. We use display to specify that we want to manage the organisation of the container's contents using the grid layout. We then use grid-template-columns to specify that we want three columns using the auto value. It is worth experimenting with adding more grid-items to your container in the HTML and adjusting the number of grid-template-columns so that you get a feel for how this works. The other properties for the grid-container are just some CSS to make things a bit prettier. The .grit-item class just contains some standard, non-grid specific CSS to make each item look presentable. Of interest here is the use of transparence in the background-color and border properties (the fourth value in each rgba value. Rather than setting the background colour of each grid-item we've set the background to white (255, 255, 255) but with 0.8 transparency which causes the background colour of the surrounding container to bleed through as a paler colour. This is a useful trick on occasion as you can just set a background colour in one place then use transparency to set of a slightly paler version of elements laid on top of the background rather than specifying the colour of each individual element.

There are multiple things that we can do to specialise the layout of items within a container, for example, getting an item to span multiple rows or columns, let's look at a simple example now:

First some HTML (grid2.html), fairly similar to before, but this time adding an individual id to each item class (so that we can treat each item individually by reference to their id).

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" href="grid2.css">
</head>
<body>

<h1>Grid Lines</h1>

<div class="grid-container">
  <div class="item1">1</div>
  <div class="item2">2</div>
  <div class="item3">3</div>
  <div class="item4">4</div>
  <div class="item5">5</div>
  <div class="item6">6</div>
  <div class="item7">7</div>
  <div class="item8">8</div>
</div>

</body>
</html>
```

Other than the change to the class names for the items, this is fairly similar to the earlier example, but with one less item.

Now some CSS (grid2.css) to style out HTML:

```
.grid-container {
  display: grid;
  grid-template-columns: auto auto auto;
  grid-gap: 10px;
  background-color: tomato;
  padding: 10px;
}

.grid-container > div {
  background-color: rgba(255, 255, 255, 0.8);
  text-align: center;
  padding: 20px 0;
  font-size: 30px;
}
```

```

.item1 {
  grid-column-start: 1;
  grid-column-end: 3;
}
```

As before we have a block for the grid-container which hasn't really changed. Now however we have an individual block for the item1 class indicating that the item should start a column 1 and end at column 3 which means that the item will span two columns. We still want to style the other items, but as we have individual ids for them, and we don't want to write a single block for every item, we've used a CSS selector '>' that selects all the direct descendents (children) of the class that it is applied to. In this case, every child of .grid-container will have the specified styles applied to them (essentially the default for all items in the container unless otherwise styled as for item1).

Finally

- W3C documentation for Flexbox: `https://www.w3schools.com/css/css3_flexbox.asp`
- Flexbox Froggy: `https://flexboxfroggy.com/` - Gamified exercises to explore the use of the CSS Flexbox
- W3C documentation for CSS Grid Layout: `https://www.w3schools.com/css/css_grid.asp`
- Gridbox Garden: `http://cssgridgarden.com/` - Gamified exercises to help learn the CSS Grid

Note: These exercises can frequently require you to do some extra reading before you have the knowledge to solve the problems posed.

Part II
Challenges

Take the plain (or lightly styled) HTML pages that you developed in previous weeks and redesign them to use different layout styles using flexbox and grid layout as appropriate. You'll probably still only need one stylesheet to account for all of the pages at this point because you want to maintain consistency between pages across the same site.

Sketch out the core kinds of page layout that you are used to seeing as a web user, for example, with header, footer, nav-bar, various amounts of columns. Try to find examples of each. Now attempt to build a page that exemplifies each layout that you've identified. The purpose of this exercise is to get you into the habit of both noticing how sites of various types are commonly laid out, and to also get you familiar with the practise of creating those same designs from scratch.

Build yourself a simple news site that display headlines and stories of topical interest. For the exercise you can source content from existing sites rather than writing it yourself. Think carefully about the layout of the pages and the logical organisation of content within each page. Perhaps different pages have different properties and readability, for example, a headlines page that displays synopsis of news stories might require a different layout to a page that tells the full and detailed story (think in terms of column width, padding, whitespace, &c.).

If you have time and there are any challenges that you haven't yet tried, give them a go then use the CSS skills that you've been buildin this week and last week to style your pages. You should now be starting to thing in terms of not only the style of individual elements, but also the overall layout and organisation of individual pages as well as site-wide design aspects. For example, how do you maintain a site-wide visual style or charatcter, whilst perhaps tuning the specific layout of individual pages to suit the content that is published on that individual page. There are not hard and fast rules here, but more of a case of buildin experience through doing this yourself, and by thinking critically about how other sites achive similar effects and design cohesion.

We will return to this topic next week to attempt to define a collection of design principles that will enable you to develop visually appealing sites without becoming a visual "designer".