Alex Bennett

Flexible Box Layout Module:

Introduction: (5min)

The need to format content in a flexible manner has never been so great. Especially when considering how many different devices are now connected to the internet.

The purpose of a *Fluid Layout* is to ensure that your website content renders in a clear, usable way on *any* resolution screen (see illustration A).

In this lesson, we will implement a fluid layout that uses responsive images in the design.(see illustration C) More on responsive images to come.

But Before we begin…

Discussion: Why the need for Fluid Layout's? (5min)

How many of you would be deterred from using a website on your phone if it's not mobile-friendly?
Would you pay to use a website that didn't look good, read well, or was hard to use?

Lesson | Converting Units of Measurement: (5min)

In the previous lesson you learned that media queries can be used to control the appearance of your website at any given browser width. When combined with media queries, layouts become a power tool to make responsive websites.

Layouts are a way of determining how to arrange your content. Since you will need to arrange your content differently on different sized screens, your layout will also need to change in relationship to the size of the viewport. In this sense, the layout of every element becomes *relative* to the browser.

The first thing we need to do when implementing a raster (pixel) based design is to convert the fixed measurement of a pixel into a relative measurement of a percentage. This will allow us to begin to think, as well as code, in terms of relative units of measurement.

The formula for converting a fixed unit of measurement into a percentage is '(target / context) * 100 = width in percent.' Another way of thinking about is take the width of element (target) and divide it by the width of the browser (context). This will work for using any fixed unit of measurement. (See illustration B)

Lesson | Flexbox intro (10min)

Great, now that we have the relative widths for all of our content, which will respond to the browser within the major breakpoints defined by our media queries, we are ready to start writing some CSS3.

Fluid layouts are usually implemented as a template, grid, or other reusable CSS system. Previous CSS technologies such as `inline block` and ` float` have fallen short in creating templates and grids robust enough to support the features often needed in addition. Luckily, CSS3 includes Flexible Box Layout Module in its official spec. https://www.w3.org/TR/css-flexbox-1/. It was introduced as the defacto solution for common CSS related layout problems such as vertical alignment.

Let's begin.

First things first, open the sample code folder, navigate to the folder called 'Starting point', open it and double click on index.html to open the website in a browser. You will see that there are 6 elements already provided. The rest of the lesson will focus on how we can use flex box to create a fluid grid with these elements.

FlexBox has four main properties: Direction, Alignment, Ordering and Flexibility. We will cover all of these properties over the lesson.

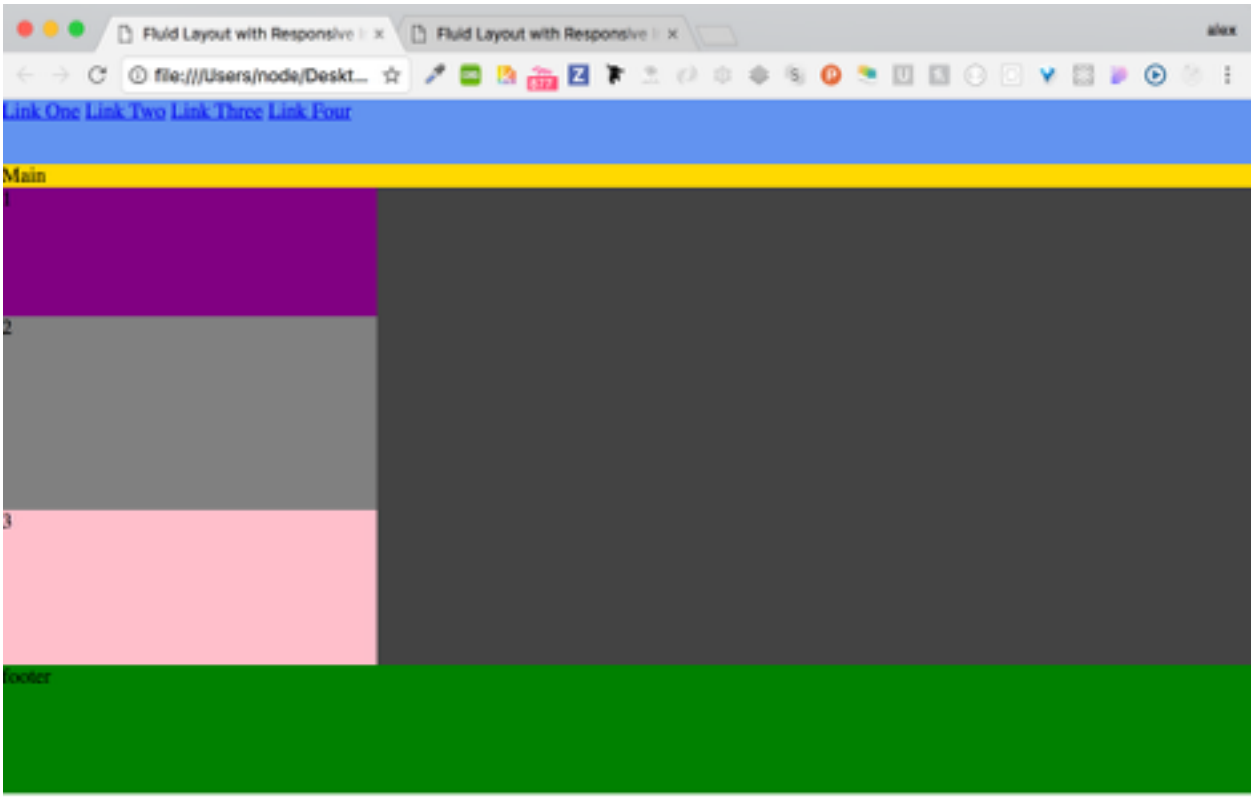
The first piece of css we need to add is on the `body` element.

```
body {
  margin: 0;
  padding: 0;
  background-color: #eaeaea;
  min-height: 100%;
  display: flex; /* this turns the entire HTML body into a flexible box module */
}
```

The key property we have defined above is `display: flex`. This is the line of code that tells the browser, "Hey, Browser treat this element like a flex box!". Awesome, we now have our first flexible box layout module.

You may notice that after this change, the content is now flowing across the page horizontally rather than vertically. This is because any element with `display: flex` on it will automatically add a property on the same element of `flex-direction`. `Flex-direction` is concerned with formatting elements in **rows and in columns.** Making it very hand for the construction of grid like systems. Let's quickly add `flex-direction: column` to the body CSS to make our elements go up and down.

Our page is looking a little bland. Lets add so width, height, and color properties to help us distinguish the elements we are working with. Take a look at illustration C to figure out approximately what relative unit of measurement each element will have. Don't forget to use the formula to convert these values.



Now that we have some basic styles. This is what you should see. Notice how the Main section is completely squished, the footer is not stuck to the bottom of the page, the three feature blocks are formatted vertically on a wide screen, and the links in the navigation are all stuck together? We will fix all of this now.

Since we already know that flex box provides a way of changing the vertical direction of content to horizontal, we can easily fix the alignment of the three feature blocks to match out prototype in illustration C.

Discussion: Does any recall the flex box property that will change the formatting of child content?
Answer: Flex-direction

Add the following bit of CSS to the .feature-set class:

```
.feature-set {
  width: 100%;
  background-color: #434343;
  // Add these lines of CSS
```

```
  display: flex;
  flex-direction: row;
}
```

— Aside Notice how we add `display:flex` to the .feature-set class? This is very important because it creates a flexible box layout module out of the .feature-set class. By doing so, all immediate children will be treated as flex box elements.

All right! Major Improvement! But now we see we have a new problem. The feature blocks are of different heights and don't look very well centered within their parent. No problem, flex box provides a solution for this. The property we are interested in using to solve this problem is `justify-content`. Justify content will allow you to center children, move them to the left, right, or even center children based on the amount of space around them. Lets take a look at this last one.

```
.feature-set {
  display: flex;
  flex-direction: row;
  width: 100%;
  height: 170px;
  background-color: #434343;
// add this line
  justify-content: space-around;
}
```

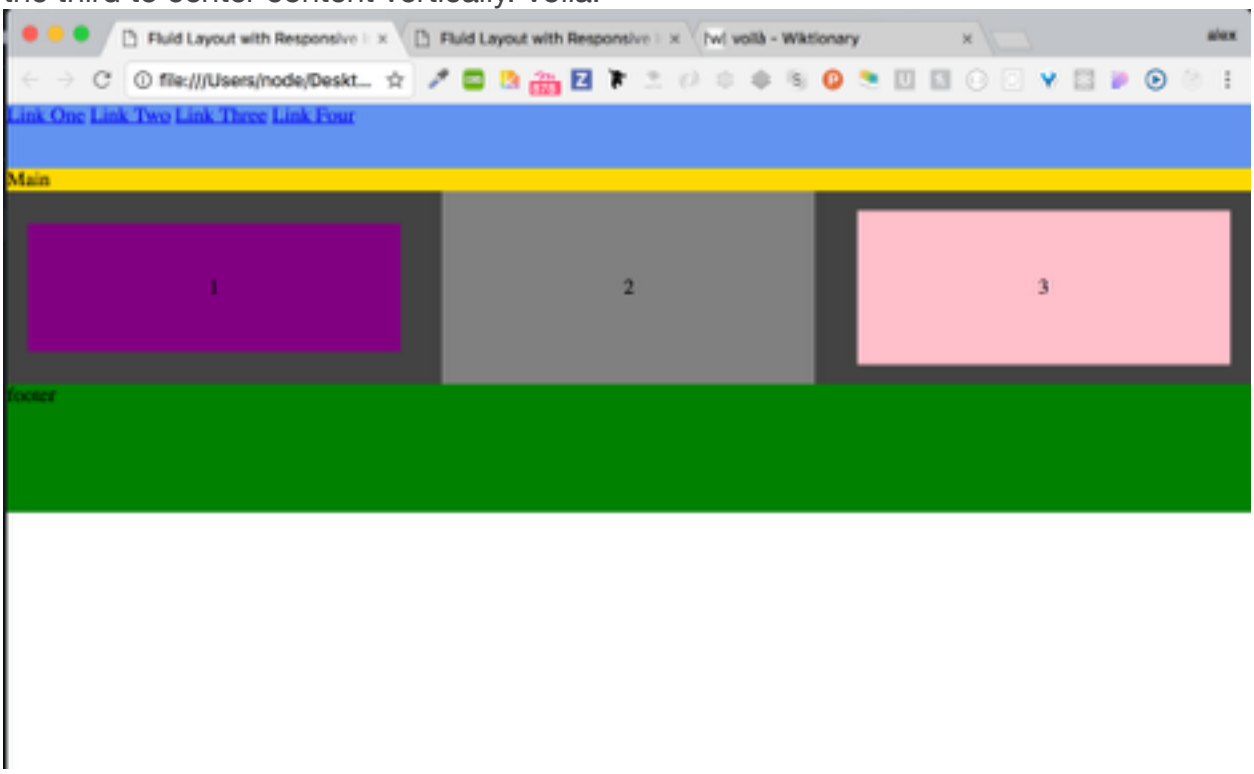
Nice job! We now have some horizontally centered elements. But, wait it still looks off. That's because we need to center this elements vertically as well. To deal with vertical alignment, flex box provides a property called `align-items`. Align items will accept the following values flex start, flex-end, center, baseline, stretch. You can play around with each one but based on our current needs, which property will me most likely use to vertically center the elements? If guessed `center` you are correct! Go ahead and add `align-items: center` to the .feature-set class.

Wow! Isn't this easy? Last thing to fix in this section. The number content in each box is also no centered vertically or horizontally. We want to fix this by turning each feature block into it's *own* flexible box layout module and to center the text both vertically and horizontally. There are only three lines of code we need to add to the `.feature` class, which cascades over each block, to accomplish this task. Can you think of the three lines of code?

That's right! We need:

```
  display: flex;
  justify-content: center;
  align-items: center;
```

We need the first to create the flex box, the second to center content horizontally, and the third to center content vertically. Voila!



Ok, moving on. You should now have something that resembles the above image.

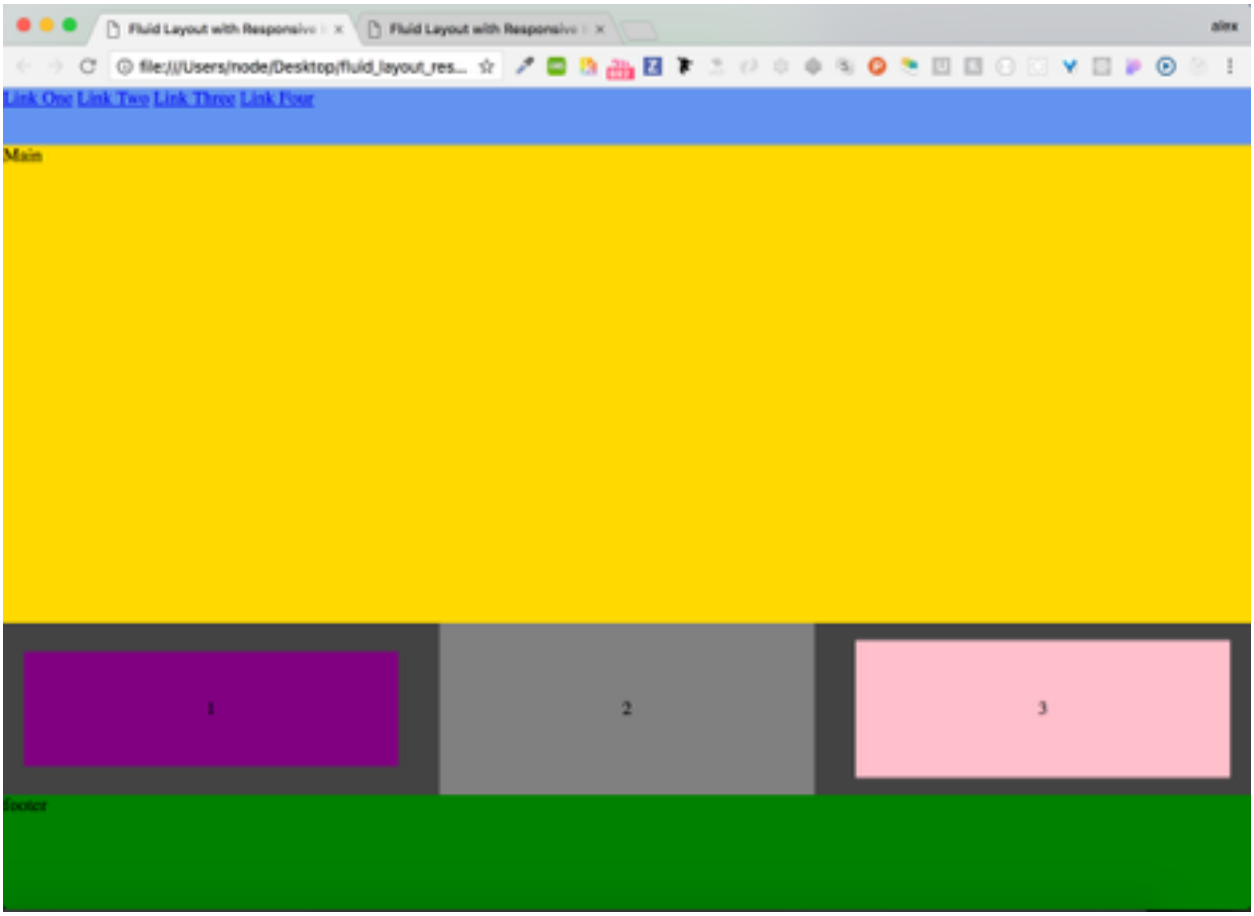
The next thing we want to work on is getting the main area to grow to be the size of a real main content area. We also want to make sure the footer lives at the very bottom of the view port at all times. We can accomplish this using one flex box property to force the main area to take up all the negative space that you currently see below the footer in the image above. This flex box property is called `flex`. The **flex** CSS property specifies how a flex item will grow or shrink so as to fit the space available in its flex container. Flex accepts three values which alter three different flex aspects of the element. Flex-grow, flex-shrink, and flex-basis. Please see reading for definitions of each.

To make our main content area take up all the negative space, we will pass the `flex` property on the main HTML element a value of 1, specifying that the main area should grow in *proportion* to the size of the negative space.

Your main HTML element's CSS should now be:

```
main {
  width: 100%;
  background-color: gold;
  flex: 1;
}
```

Boom! See how that change also pushed the footer down to the bottom of the page? Lovely! This is what you should be seeing:



Notice that there is no longer and white or negative space on the page.

For our next few changes, let's focus on the navigation. What we want is to have one link to the left and three links to the right. Flexbox also offers a way we can use to implement this design. First things first. We need to make the nav HTML element a flex box with the links vertically aligned. Can anyone tell me what two lines of code we need to do this? Add `display: flex` and `align-items: center` to the `nav` HTML element CSS.

Ok now for some magic to get the last link all the way to the right side of the navigation. Because we are wrapping the navigation links in a flex container, each link is now a `flex-item`, which get laid out using the flexbox model. Because the flex box model is different, when we target the last CSS link using `.link-item:last-child`, all we

need to do is add `margin-left: auto`. This will force the last flex item to take up the full available left side margin.

add the following CSS to your styles.css file:

```
nav {
  width: 100%;
  height: 50px;
  background-color: cornflowerblue;
  display: flex;
  align-items: center;
}

.link-item {
  margin-right: 1em;
}

.link-item:last-child {
  margin-left: auto;
}
```

notice that we treat the last item differently.

you should now see something close to the above image.

Finally, we need to address out layout on mobile. To do this we will utilize a simple media query that targets any screen size smaller than 800px. For this design, we want the 3 feature blocks to render vertically and to also *switch* their order.

To do this we will utilize two flex box properties. Flex-direction and order. Go ahead and add the following media query to your styles.css file:

```
@media(max-width: 900px) {

}
```

the first thing we will do make sure the blocks render vertically using `flex-direction: column;`.
 you should now have:

```
@media(max-width: 900px) {
  .feature-set {
    flex-direction: column;
    height: 300px; // go ahead and add this for effect
```

```
  }
}
```

Secondly, we want the blocks to read 3, 2, 1. To do this, we will utilize the order property. We need to manually set the order on each of our three elements by adding the property `order: <number>`. So, we will now add the following CSS to the media query:

```
  .feature-1 {
    order: 3;
  }

  .feature-2 {
    order: 2;
  }

  .feature-3 {
    order: 1;
  }
```

let's make the blocks take up the width of the device, using our relative units of measurement. Please add to your media query:

```
  .feature {
    width: 98%;
    height: 90px;
  }
```

Your final media query should now look like:

```
@media(max-width: 900px) {

  .feature-set {
    flex-direction: column;
    height: 300px;
  }

  .feature {
    width: 98%;
    height: 90px;
  }

  .feature-1 {
    order: 3;
  }
```

```
  .feature-2 {
    order: 2;
  }

  .feature-3 {
    order: 1;
  }

}
```

More on responsive Images … Did not get to finish this section!

Open the ‘final’ folder in the sample code and double click on index.html to see what you website should look like.

Lesson summary: In this lesson you learned how to create a fluid layout with responsive images.