UNIT-III

FRAMEWORKS IN RESPONSIVE DESIGN

GRID SYSTEMS:

- A grid system is a structure within which you can build the layout of your web site.
- A grid system can offer significant benefits when building a site, allowing you to achieve consistency of spacing in your build, with which comes improved usability of the site because the text is easier to follow.
- Despite the benefits of mobile first responsive design and a drive within the web development community to embrace it, a majority of standalone grid systems are still built desktop first.
- Therefore, when looking at grid systems, it is important to look at whether they are built to be mobile first.

There are three of popular mobile-first responsive grids. The grid systems examined are:

1. Fluidable
2. CSS Smart Grid
3. csswizardry-grids

Fluidable:

- Fluidable is a mobile-first, responsive grid system developed by Andri Sigurðsson, which has fixed gutters and supports variable column widths.
- It is possible to see in fluidable, capability to define how many columns the element spans for each different device type, meaning that when using this technology it is possible to easily control how the site looks down to an individual device type.

The way this is achieved is through the naming conventions used for the elements’ class names:

1. .col-mb-x Defines how many columns it should span on mobile
2. .col-x Defines how many columns it should span on a tablet
3. .col-dt-x Defines how many columns it should span on a desktop

For each of these, the x represents the number of columns the element should span.

CSS Smart Grid:

- The next grid system is CSS Smart Grid by Daniel Ryan.
- The columns in CSS Smart Grid are defined by adding the class “columns” with an additional class to specify the number of columns it should span (e.g., one, two, three).
- This is supported right up to 12 columns. Alternatively, you can also use keywords to specify how wide an element should be, these are:
    1. one-fourth: equal to 25 percent or three columns’ width
    2. one-half: equal to 50 percent or six columns’ width
    3. one-third: equal to 33.33 percent or four columns’ width
    4. two-thirds: equal to 66.66 percent or eight columns’ width

An example of using the grid in practice is shown below. As you will see, a row is defined, along with two columns, each with a width defined using the keywords discussed above:

```
<div class=”row”

        <div class=”colums two-thirds”>

              <p>Lorem ipsum</p>

         </div>

         <div class=”columns one-third”>

              <p>Lorem ipsum</p>

          </div>

</div>
```

Using these keywords adds extra flexibility to how you choose to label your columns, however, it is important to note that in the CSS it is doing the same as if you had simply used the numerical column class names.

csswizardry-grids:

The next grid is the csswizardry-grids by Harry Roberts .It is a mobile-first responsive grid available for download on GitHub.

Similar to the CSS Smart Grid, this grid system uses a class-naming convention that refers to the column widths as fractions, examples being:

1. one-half
2. one-third, two thirds
3. one-quarter, two-quarters, three-quarters .

This class-naming convention lends itself to being easier to read than the common span-X approach used by a large number of grid systems. Again, in a similar way to CSS Smart Grid, csswizardry-grids allows you to use SASS with it and you can simply extend the rules for the columns to your own CSS class selector, meaning that if you choose, you can avoid adding the extra classes into the HTML itself.

One of the main disadvantages of using csswizardry-grids is that rather than using floats, it uses inline blocks to place the columns next to one another, so if you have a space between the columns in your HTML, you get unwanted spacing in your rendered page. There are two common ways you can use to prevent this: either by placing an HTML comment in between the elements,

```
<div class=”one half”>

         Lorem ipsum

</div><!—
```

```
--><div class=”one-half”>
        Lorem ipsum
</div>
```

Or by simply removing the spaces in between your columns,

```
<div class=”one-half”
      Lorem ipsum
</div><div class=”one-half”>
      Lorem ipsum
</div>
```

CSS FRAMEWORKS:

A CSS framework is an extension of a typical CSS grid system in that in addition to providing the grid layout, it also provides standard browser resets, typography, and user interface elements that you can use to build responsive sites.

Although it is called a CSS framework, it is important to be aware that some of the interface elements common in these CSS frameworks often require JavaScript, which ships with the framework.

There are many different CSS frameworks available, however, there are two that really have been embraced in the web development community: **Twitter Bootstrap and Zurb Foundation**.

Common user interface elements that you will find in a CSS framework are:

1. Dropdowns
2. Button’s and button groups
3. Forms
4. Navbar
5. Breadcrumbs
6. Pagination
7. Labels
8. Badges
9. Page header
10. Thumbnails
11. Alerts
12. Progress bars
13. Media object
14. List group

Through providing a wide selection of components, CSS framework makes the life of the developer easier as they can simply choose from existing components when they need a common element, allowing them to focus their time on the look and feel of the site along with the new custom components that are adapted to the site they are building.

Twitter Bootstrap :

Twitter Bootstrap is a CSS framework that was initially built by Mark Otto and Jacob Thornton; however, it has since become the most popular project on GitHub with almost 600 contributors and over 25,000 forks. The framework is also quite mature, having already had 25 releases, bringing it to version 3.2.0 at the time of writing.

Examples of some of the popular add-on components are:

1. Fuel UX: Fuel UX extends Bootstrap by adding additional lightweight JavaScript components.

2. Bootstrap Image Gallery: This component adds the ability to have a gallery with a light box that can navigate through a series of images.

3. Bootstrap Application Wizard: The Bootstrap Application Wizard component allows you easily to make multipart forms with Bootstrap.

Zurb Foundation:

Zurb Foundation is a CSS framework built by the web design agency Zurb, first becoming open source as of version 2.0 in 2011 .

The components in Zurb Foundation are very similar to those in Twitter Bootstrap.

When you are building HTML you do not need to include the HTML for all of the viewport widths in the main page, instead focusing on providing mobile-first HTML and then using Interchange to load the additional HTML for different viewports. Additionally, Interchange can be used to load different images based on the width of the viewport, so you can load smaller, optimized images on smaller viewports and larger images for larger viewports. Zurb Foundation is not just an open source project that Zurb develops.

WHAT IS BEST FOR MY SITE?

It is important to make the decision of which framework or grid system you want to use, if you use one at all, when you initially start to build a site as it can be very difficult to change once you have already progressed with your build.

Choosing a CSS Grid:

Having decided that a grid is more your cup of tea and you want to be able to build all your own components, it is important to take a look at the options available: When choosing a grid, there are some questions you will need to ask yourself:

1. **Do you feel comfortable working with the grid?:** It is important that you feel comfortable with the grid, even small things like how classes are named can be an annoyance if it does not fit with the way you normally code, so it is important that you are 100 percent comfortable with the grid, especially because you may be maintaining the site for a number of years.
2. **Is the grid suitable for your site?:** Your grid of choice should work for your site. This is especially important if you already have had the site designed without a prior discussion about grids. This means if you want to use a grid, it needs to fit the number of columns for which the site has been designed. Some grid systems will allow you to configure the number of columns they use, which can aid you in trying to get a design to fit into a grid. A better approach is to have chosen the grid before the site is designed and discuss this with the designer so you know your grid is suitable.

Choosing a CSS Framework

When choosing a framework, there are some questions you will need to ask yourself:

1. Do you like the grid the framework is built on?: As previously discussed, the base of a CSS framework includes a grid system, so you have to be comfortable with the grid.

2. Are you happy with the selection of components that the framework comes with?: The user interface components that the framework comes with are a core part of the framework, so it is important that you are happy with the selection. Although you can add your own, using the components that come with the framework will not only make your life easier it can also save you a significant amount of time.

3. Does the framework suit the way you work and the tools you use?: Some frameworks might require you to use specific tools, and it is likely you already have a set of tools you know and love, so you will need to consider this when choosing a framework.

Choosing Neither a Grid Nor a Framework

There are also some instances where neither a CSS grid system nor a CSS framework is appropriate, examples of these being:

1. The design uses uneven column sizes.

2. The design has uneven margins in between the columns.

3. The width of the design is not easily divisible.

4. The site content is placed on the page in a irregular manner (an example of this could be a parallax site).

PROTOTYPING A SITE USING A CSS FRAMEWORK

CSS frameworks really shine for being able to quickly prototype a web site using the components provided as part of the framework. Without writing a single line of CSS, it is very easy to quickly mock up prototypes using the included components.

You need to set up a base template for the components to sit within. For this, you will need to include the Twitter Bootstrap CSS framework.

Basic template HTML:

```
<!DOCTYPE html>
<html>
<head>
        <title>Prototype</title>
        <meta name=”viewport” content=”width=device-width”>
        <link rel=”Stylesheet” type=”href=”css/bootstrap.min.css”>
</head>
<body>
        <script src=”http://code.jquery.com/jquery-1.10.1.min.js”></script>
        <script src=”js/bootstrap.min.js”></script>
</body>
</html>
```

With this in place, you can now start building the prototype out of the Twitter bootstrap components.

Without writing any CSS and simply adapting the HTML found in the documentation, it is really easy to create a prototype using a CSS framework. Although this is a very simple example, it really highlights some of the benefits CSS frameworks have to offer. There are several key benefits of building a prototype rather than spending time creating wireframes:

1. Those using the prototype can get a feel for how the site will work rather than relying on the notes attached to the wireframe.

2. By using a responsive CSS framework, your prototype is by default responsive, allowing you to show how the site will function on different devices.

3. Rather than starting from scratch when building the site, you might be able to build on top of the prototype.

ADAPT AN EXISTING SITE

ADAPT EXISTING STYLES AND SCRIPTS

The first of the three approaches for adapting a site is to keep the existing HTML and to adapt the existing styles and JavaScript so they respond to the device viewport by adding media queries.

There are many ways to degrade the content of a site, the first of which is to simply hide the less important content and rearrange the remaining content so it fits nicely on the smaller device’s screen, both of which can be achieved using CSS alone.

This approach has the benefit that it can be done relatively quickly, however, it is possible that content you deem to be less important may be important to one or more of your users. The second approach would be to adapt the content using CSS to work better on the site, and where appropriate to use JavaScript to change the way the user interacts with the content.

Defining the Breakpoints

A set of breakpoints that will allow it to work across a wide variety of devices.

For the first breakpoint, I want to try to scale down the existing site so it fits comfortably on small devices. I will want to add a breakpoint so that when the existing site no longer fits within the viewport, it can switch to the smaller version of the site.

The code for this breakpoint is:

```
@media screen and (max-width: 1000px){

  }
```

The next breakpoint will need to target mobile devices.

you can set the breakpoint for mobile devices to 767px, as this is 1px less than tablet. It is the extra small breakpoint.

The code for this breakpoint is:

```
@media screen and (max-width: 767px){

 }
```

Typography

The existing typography is a good size for the majority of devices, however, on extra small devices it is a bit too big, so you will need to add some styles to your headings. To do so, you need to look at the base font size that has been used so you can calculate the font size values correctly.

The code for this would be:

```
 @media screen and (max-width: 767px){
  h1
     {
          font-size: 22px;
          font-size: 1.571em;
     }
 h2
   {
         font-size: 18px;
         font-size: 1.286em;
   }
}
```

Wrapper

Having adjusted the font sizes, you now need to adjust the width of the wrapper so that is does not extend farther than the bounds of the devices viewport. To do this, you need to decide how your site should work across the different breakpoints you have defined. The first breakpoint is the small breakpoint, which is defined as having a max-width of 1000px. For this breakpoint, you would set the width of the wrapper for the site to 710px, as the site needs to work on viewports ranging from 768px to 1000px, and this includes scrollbars. You also need to take into account the 20px left and right paddings.

The code for this breakpoint would be:

```
  @media screen and (max-width: 1000px){
      .wrapper{
      width: 710px;
        }
   }
```

The final styles for the wrapper for the extra small viewport would be:

```
 @media screen and (max-width: 767px){
     .wrapper{
            width: 100%;
            -moz-box-sizing: border-box;

            -webkit-box-sizing: border-box;

            box-sizing: border-box;

        }

  }
```

Jumbotron

The Jumbotron currently is expanding outside the site's wrapper. If you look at the existing CSS for the Jumbotron to see why this is happening, you will see that the Jumbotron has its width set to 940px. Clearly the media queries need to be adjusted to how the Jumbotron looks across different devices.

If the heading text does not approach the edges of the Jumbotron, you would add padding to the heading text.

```
   @media screen and (max-width: 767px){

        .jumbotron h2{

            font-size: 22px;

            font-size: 1.375em;

            padding: 10px 20px;

       }

}
```

Products

The next step is to adjust the products so they look great across the various breakpoints. First, you would start with deciding how they should look on small devices. To determine this, you need to consider whether the existing content will fit comfortably within the existing structure or whether you will need to adapt the layout. Adapting the layout could be as simple as changing from a three-column layout to a two-column layout, or it could mean changing the layout to be single columns with a fluid width. In the case of this example, however, on the small breakpoint the existing layout and content can fit comfortably within the existing three-column structure by simply changing the width of the three product elements.

The code for this:

```
    @media screen and (max-width: 767px){

       .product:last-child{

            padding-bottom: 0px;

         }

     }
```

REFACTORING

The second approach that could be taken for adapting an existing site is refactoring the existing code base to be mobile first. The benefit here is that you are not having to start the styles from scratch.

Defining the Breakpoints

As the first step of the refactoring of the site, you need to wrap the existing styles that are currently being used to style the site in a media query targeted toward the larger devices, such as desktop browsers that the site already supports. To choose a suitable media query for this, look at the site's existing styles to determine its width.

```
@media screen and (min-width: 1024px){

}


@media screen and (min-width: 1200px){

}


@media screen and (min-width: 768px){

}
```

The breakpoints can be changed according to the devices.

Refactoring

The Existing Styles Having defined the breakpoints, you can now refactor the existing styles. You have moved all the existing styles into a breakpoint.

The process of refactoring is with the site's main wrapper. The padding and margins that are applied can easily be pull outside the media query. The other styles are specific to the larger viewport sizes.

```
 .wrapper{

     padding: 0 20px 20px;

     margin: 0 auto;

  }
```

Header

The next step is to look at the header styles to see which styles can be used for smaller devices. In this example, the existing styles applied to the header are simply spacing and a border bottom. It makes sense to move these out of the media query so they are accessible to the other viewport sizes.

Jumbotron

The next element of this site that needs to be looked at is the Jumbotron. The original CSS for the Jumbotron is 35 lines long, and much of it applies to only the larger viewport widths, so it is important to look carefully at what is suitable.

```
.jumbotron{
    background: #f4a156;

    margin: 0 -20px 20px;

    padding: 20px;

    text-align: center;

}
```

The font sizes are also particularly large. You also don’t want additional spacing above the second-level heading in the Jumbotron, so specifically for this Jumbotron that the margin top is set to 0px.

```
h2{

 font-size: 1.3em;

 }

 .jumbotron h2{

 margin-top: 0px;

}
```

Product Panels

The next step is to build the styles for the product panels. The majority of the styles used for these panels on the larger viewports can also be used to style these panels on the smaller viewports.

Add some extra padding and borders to finish off the product panels. So that there is spacing on either side of the border, add 20px padding to the top and bottom of each product panel. The border is then applied to the bottom of the product panel.

the CSS that is currently applied to the footer in the larger media query.

Targeting the Different Breakpoints

At this point the site is working great on extra small devices, but we haven’t yet thought about how it looks on small or large devices. It is also likely we have broken the original (medium) layout due to the new styles that have been added.

If there is a lot of space around the stacked product panels that could be better utilized by showing the products in a row next to one another, set a fixed width to the product panels, you would apply a fluid width. Because you want three columns here, you would set the width to 33.33%. To get the panels to sit next to one another, apply a float and remove the padding from the top of the elements. The final code looks like this:

```
.product{

   width: 33.33%;

   padding-top: 0px;

   float: left;

    }
```

FULL RESKIN

There are benefits and disadvantages of choosing to take the full reskin approach, with the key benefit being that by starting the CSS afresh and refactoring the HTML as necessary, you overcome any inefficiencies that may have creeped into the CSS over the life of the site. In addition, because you would be replacing the existing CSS, you can take a mobile-first, progressive-enhancement approach, building the site for the smaller, mobile devices first and then enhancing the site based on the devices feature set and viewport size.

Preparation

Being a full reskin, the first step is to clear out the original styles and see what is left. By deleting the core styles, you are left with simply the content.

Part of the reason that this remains very easy to follow is that the heading tags were used correctly, so while the site has only browser default styling, the content still has a strong, clear heading hierarchy. When looking at your own site, you might notice that some of your headings do not seem to follow a proper hierarchy. This is a good opportunity to look at refactoring your headings so they are used to provide users with a proper hierarchy to the page.

General Styles

Before starting the core of the coding, let's first change the default box model for the elements. This is achieved using the box-sizing CSS3 property, and applying it to all elements using the CSS universal selector (*):

```
* {
    -webkit-box-sizing: border-box;   /* Safari/Chrome, other WebKit */
    -moz-box-sizing: border-box;      /* Firefox, other Gecko */
    box-sizing: border-box;           /* Opera/IE 8+ */
 }
```

Defining the Breakpoints

The first breakpoint to define is a minimum width of 768px. This will allow you to target devices with a viewport width starting at 768px. The reason behind this choice is that this is normally the starting point for tablets in a portrait orientation, and as tablets get bigger screens and wider viewports, we can then enhance the site to take advantage of this.

```
 @media screen and (min-width: 768px){
 }
```

The second breakpoint to define is a minimum width of 1000px, the purpose being to target the very popular 1024×800 screen resolution used by a large number of users. Although the previous breakpoint could assume the majority of users would be using some kind of tablet, this breakpoint is shared between landscape tablets and desktops, so it can't be used to make these kinds of assumptions.

```
 @media screen and (min-width: 1000px){
 }
```

The final breakpoint to define is a minimum width of 1200px. With the growth of larger displays, as discussed earlier in the book, it makes sense to take advantage of the additional space available.

```
@media screen and (min-width: 1200px){

 }
```

Typography

Once we are happy with the breakpoints, let’s start by applying sensible default sizes to the typography. The first step is to set the base font size for the page. For this, a sensible default size is 14px, and this is set on the HTML tag as a percentage:

```
html{

        font-size: 87.5%;

 }
```

Setting a default font size will have an effect on all the text on the site, however, you may want to set your own custom font sizes for the headings:

```
 h1{

          font-size: 22px;

          font-size: 1.571rem;

  }

 h2{

         font-size: 18px;

         font-size: 1.286rem;

  }
```

Wrapper

As you we already know, the HTML includes a div element with the class wrapper, which encapsulates the HTML of the site. In the site’s original CSS that is being replaced, this was used to define the width of the site and center it on the page. For a mobile device, however, we would not want to set a width because we want the wrapper to be fluid. We do, however, want to apply consistent spacing to the sides of the site, so we can use a wrapper for this.

Header

Having looked at both the typography of the site and the wrapper, let’s move on to styling the header of the site. On extra small devices, we want to have a border on the bottom of the header spanning the width of the viewport. However, with a set margin of 20px for the wrapper, the header does not naturally span 100% of the viewport. To fix this, let’s use negative margins to pull the width of the header to the full width of the viewport. You can then add this spacing onto the inside of the header using padding and then add the border to the bottom of the header.

Jumbotron

Directly under the header is the Jumbotron, it is used to highlight the latest product that our fictional company Unresponsive design inc. wants to promote. This panel should stand out, so we add a background color of orange, along with pulling out the Jumbotron so it fits the full width of the viewport.

Products

The products are a core part of this site, so it is important to show them in a way that tells the user about them in a clear and concise way. There is already HTML that lists the price for the name and the specification of the product, however, you would need to style this so it works well on extra small devices. It makes sense that on these devices the products would be stacked so the user can scroll up and down to see the different products. As each product is a div, which is a block-level element, each would naturally be full width and stacked. But it is also important to apply styles to space out the products. Finally, you would align the text to center to be consistent with how the header and Jumbotron are styled.

Footer

The final thing to style for the reskin is the footer. This is really simple because the products already provide a separator between the products and the footer, so all you need to do is center align the text and add a tiny bit of spacing below:

```
.global-footer{
    text-align: center;
    padding-bottom: 10px;
  }
```