UNIT – II

The Power of Media Queries

An Introduction to Media Queries

Media queries form the foundation of responsive design, allowing you to change the look and feel of the web site based on a query you have specified. A media query can be split into two types of components: media types and media expressions.

Media Types Prior to the new CSS3 draft specification, CSS2.1 introduced media types, which allow developers to add media-dependent style sheets targeted toward different types of devices. There are ten different media types in total, three of which you are most likely to already have encountered: all, screen, and print. If you have encountered media types in the past, it is likely you have seen them used to enable or disable style sheets based on a device type. An example of this is:

```
<link rel="stylesheet" type="text/css" href="all.css" media="all" />

<link rel="stylesheet" type="text/css" href="screen.css" media="screen"/>

<link rel="stylesheet" type="text/css" href="print.css" media="print"/>
```

In addition to screen and print style sheets, there are many other media types that enable support for a wide variety of other devices:

1. *all*: all devices
2. *aural*: speech synthesizers
3. *braille*: braille tactile feedback devices
4. *embossed*: paged braille printers
5. *handheld*: small or handheld devices
6. *print*: printers
7. *projection*: projected presentations, like projectors and projected slides
8. *screen*: computer screens
9. *tty*: media using a fixed-pitch character grid, like teletypes and terminals
10. *tv*: television-type devices

Aside from being able to specify a different style sheet for each media type, it is also possible to include these styles in your style sheet by using the following syntax:

```
@media print
{
    /* print styles go here */
}
```

There are a couple of benefits from including the different media type queries within your main style sheet:

1. It reduces the number of HTTP requests. An increase in the number of HTTP requests can lead to reduced performance of loading a site.
2. Separate style sheets can increase the time that the page is blocked from rendering, as most browsers will wait until all the individual style sheets are downloaded before they will render the content.

Media Queries

Media queries were added in CSS3 as an extension of media types with the aim of giving developers more control over how their web sites are displayed on different browsers and devices. The idea is that rather than having to build and maintain multiple versions of each page for different devices, you can instead adapt a single web site in your CSS based on the attributes, properties, or characteristics of the device.

Unlike media types that simply tell you the type of the device, media queries add a level of logic to CSS that says if a condition is met, then the styles should be applied, otherwise they should be ignored. This means that rather than simply targeting a device by type, you can now target the individual characteristics of the device.

What Can Media Queries Test for?

There are a variety of different types of query for which a media query can test. These are discussed in the sections that follow.

width | min-width | max-width

The width query allows you to test against the width of the browsers viewport. This enables you to target styles at specific browser widths. Not only are you able to test against a set width, but you are also able to target either the minimum width or maximum width of the browser viewport. This means you can use this query to match a wide variety of different device widths. The width media expression is one of the most used media expressions for adapting sites to be responsive.

height | min-height | max-height

The height query allows you to test against the height of the browsers viewport. Similar to the width query, you are able to target an exact height, minimum height, or maximum height. Although the height query is used less often than the width query, the height query can be especially useful where you want to ensure specific content is viewable (or not viewable) to your users when the page first loads, as you can use it to adjust the height of your content to best suit the height of the viewport.

device-width | min-device-width | max-device-width

The device-width query allows you to test against the width of the device. You are able to target either an exact width, minimum width, or maximum width. The difference between width and device-width is that width is related to the width of the browser, whereas device-width is related to the width of the screen of the device. Although there are some use cases for using device-width, the problem is that if the user resizes the browser on a desktop, the site wouldn't resize to fit. Additionally, if you are using the viewport meta tag and setting width to be equal to device-width, you should just use the width query.

device-height | min-device-height | max-device-height

The device-height query allows you to test against the height of the device. You are able to target an exact height, minimum height, or maximum height. The difference between device-height and height is that device-height relates to the height of the device's screen, whereas height relates to the height of the viewport. This distinction is important on devices where you are able to resize the browser window.

aspect-ratio | min-aspect-ratio | max-aspect-ratio

The aspect-ratio query allows you to test against the aspect ratio of the device's viewport. The aspect ratio of a device is the ratio between the length of the longer side of the device vs. the length of the shorter side of the device. The aspect-ratio query can be especially useful when you want to target assets to match a device's aspect ratio, including showing video, which is best optimized to the user's device.

device-aspect-ratio | min-device-aspect-ratio | max-device-aspect-ratio

The device-aspect-ratio query allows you to test against the aspect ratio of the device. The difference between device-aspect-ratio and aspect-ratio is that device-aspect-ratio relates to the aspect ratio of the device's screen, whereas aspect-ratio relates to the aspect ratio of the viewport. This distinction is important on devices where you are able to resize the browser window, as the aspect-ratio is as fluid as the size of the viewport, whereas the value for device-aspect-ratio will not change.

color | min-color | max-color

The color query allows you to test against the color capabilities of the device based on the numbers of bits per color components.

color-index | min-color-index | max-color-index

The color-index query allows you to test against the number of colors a device supports, and the value must be an integer and cannot be negative.

monochrome | min-monochrome | max-monochrome

The monochrome query allows you to test against the bits per pixel on a monochrome device, which uses 1 for true and 0 for false.

resolution | min-resolution | max-resolution

The resolution query allows you to test against the pixel density of the device. The resolution query accepts three different types of values: dpi (dots per CSS inch), dpcm (dots per CSS centimeter), and dppx (dots per pixel). The preferred option is to use dppx, which was a more recent addition to the specification than both dpcm and dpi.

The benefit of dppx over its predecessors is that it is directly related to pixel density of the screen so it is much easier for developers to understand.

scan

The scan query allows you to test against the scanning process of a device. This is very specific to televisions, which can have progressive or interlace scanning. The difference between these is that a *progressive* display draws all the lines on the display at once, and an *interlace* display draws all the odd lines, then draw the even lines, to trick the eyes into thinking they are seeing all the lines at once.

grid

The grid query allows you to test whether the device is a grid device or a bitmap device, with two possible values. If the value is set to 1, the query will enable the CSS if the device's display is grid based, an example being a phone display with only one fixed font. Alternatively, you can check for all other devices by setting the value to 0.

orientation

The orientation query allows you to test whether the device is landscape or portrait and apply your CSS appropriately. A typical use case for the orientation query is where you might want to switch between a single column in portrait to two columns on landscape.

Syntax of a Media Query

Now that you have familiarized yourself with media types and media queries and their uses, let's explore the methodology of writing them.

A media query is made up of at least the media type and can additionally have one or more media expressions, which return either true or false. For the CSS to be applied, the media type should match the device the page is loaded on and all the media expressions must return true. The media queries can be as specific or ambiguous as you want, allowing you to ensure your CSS is applied exactly in the way you expect. The best way to look at the syntax of writing media queries is to dive into some examples.

The first example looks at how you can add CSS to devices with a small viewport; typically a mobile phone. If you want to only enable your CSS on a small viewport, you need to add a rule that defines an upper limit for the width of the viewport the CSS will be applied to. In this

case, you would add a max-width rule and set the value to 767px. A simple example of this would be:

all and (max-width: 767px)

Media Type — Media Expression

In this example, the media type is set to all, which means that it applies to all the media types. There is then a media expression with max-width set to 767px, because you want to target extra small devices (e.g., mobile phones), which in this case is defined as less than 767px. The reasoning here is that many of the devices that might be classified as small devices, such as the iPad, have a minimum width of 768px. The logic behind this media query says that for all devices, if the width of the viewport is less than or equal to 767px, the CSS styles are applied.

If you only wanted to apply the CSS to screen devices, you could change the media type used in the media query to screen. This change is reflected in this updated example:

screen and (max-width: 767px)

Media Type — Media Expression

The logic to this media query is similar to that of the previous example, however, the key difference is that you have changed the media type to screen. The logic, therefore, is that for all devices of the type screen, if the width of the viewport is less than or equal to 767px, then the CSS is active.

The "Not" Logical Operator

If you wanted to apply CSS to all media types other than screen, you can take advantage of the not logical operator. The not logical operator in media queries tells the browser to reverse the results of the expression. With the screen example, you can simply add the not logical operator to the beginning of the expression:

not screen and (max-width: 767px)

Media Type — Media Expression

So for this example, the logic is that if the browser is not a screen device with a width less than or equal to 767px, then the style sheet will be activated.

The "Only" Logical Operator

The only logical operator is used to prevent an older browser that does not support media expressions from trying to process the media query. Without the only operator, the older browser will read the media type, however, it will not understand the media expressions; therefore, the media expressions are ignored and the styles are applied.

The reason for this behavior is that the media attribute was originally used for media types prior to being used for media queries. Although the CSS specification has extended to include media queries in CSS3, you still need to support older browsers that use the older specification. A media query using the only logical operator simply prepends the media type operator with only. This example updates the earlier media query for testing for extra small devices:

only screen and (max-width: 767px)

Media Type

Media Expression

The logic to the media query in this example is exactly the same as before, however, you have now added the only logical operator, which will prevent the styles wrapped in this media query from being applied incorrectly in older browsers.

Using Multiple Expressions

One of the things that makes media queries really powerful is the ability to use multiple media expressions together. What this means is if you want to target small devices, like tablets, without affecting the extra small devices, you can do so, using two media expressions rather than one.

To chain the expressions, you need to use the and keyword in between the media expressions. This example uses a min-width of 768px and a max-width of 1023px to determine whether the device is a small device:

screen and (min-width: 768px) and (max-width: 1023px)

Media Type

Media Expression

Media Expression

The logic for this media query is that if the media type is screen, the viewport is equal to or greater than 768px, and is less than or equal to 1023px, then the style sheet will be activated.

Chaining Media Queries

So far the examples have simply been using individual media queries; however, by chaining the media queries and allowing the CSS to be applied in multiple circumstances, you create the benefit of CSS being applied when any of the queries return true. Each media query can have its own media type and its own media expressions, meaning that the separate media queries can target different media features, types, and states. To use multiple media queries, simply separate them by adding a comma in between each query and then each of queries will be individually assessed to see if they are true, as shown in this example:

```
screen and (min-width: 768px) and (max-width: 1023px), screen and (orientation: portrait) { ... }
```

Media Type | Media Expression | Media Expression | Media Type | Media Expression

In this example, there are two media queries. The first media query will be true if the width of the browser is equal to or greater than 768px and is less than or equal to 1023px. The second media query will be true if the orientation of the device is portrait. Now if either of these two media queries is true, then the styles will be activated.

A common example of where you might want to use multiple media queries is to target a feature that is implemented differently in different browsers. An example of this is when you are using the min-resolution expression, as shown in this code:

```
@media only screen and ( -webkit-min-device-pixel-ratio: 2 ),

only screen and ( -o-min-device-pixel-ratio: 2/1 ),

only screen and ( min--moz-device-pixel-ratio: 2 ),

only screen and ( min-device-pixel-ratio: 2 ),

only screen and ( min-resolution: 192dpi ),

only screen and ( min-resolution: 2dppx ) {
```

```
  /* High resolution styles here */
}
```

In the early days of CSS3 media queries, there was the need to implement a way to target high pixel density devices, which lead to WebKit implementing the device-pixel-ratio expression under their vendor prefix. As the specification has matured, the correct implementation is to use a “resolution” expression; however, to support these older browsers, you need to use multiple media queries.

Using Media Queries in CSS

Now that you have explored the syntax of writing media queries, it is time to look at how you can apply them to a page.

The three methods are:

1. Separate style sheets for each media query
2. Use @import in the main CSS file to load CSS files conditionally
3. Use the media queries inside CSS file

Separate Style Sheets

Typically when using screen or print style sheets in the past, there was a style sheet for each with the rules of which to apply to which media type. Similarly, you can do this with media queries to enable and disable a style sheet conditionally. To do this you can use the media attribute on the link tag used to include the style sheet. This is the same attribute you may have previously used to add a media type rule to your style sheet. When adding the rule, simply drop it into the media attribute, as shown in this example:

```
<link rel="stylesheet" media="screen and (min-width: 768px) and (max-width: 1023px)
" href="tablet.css" />
```

In this example the media query is taken for checking whether the user is on a small device and this is simply added to the media attribute.

The benefit of using separate style sheets is that it allows you to compartmentalize code easier. Please bear in mind that all style sheets in a media query are downloaded to the user’s device and are simply not active if unneeded. This means that by using separate style sheets you are increasing the number of HTTP requests made to the server.

Use @import

If you want to use separate style sheets for each media query but do not want to have to define them in the HTML, you can use the CSS @import syntax with a media query applied. The @import syntax can be broken down into three parts:

1. The @import declaration
2. The URL to CSS file to include
3. One or more media queries

Putting this together, you end up with the following:

```
@import url("tablet.css") screen and (min-width: 768px) and (max-width: 1023px);
```

In this example you have taken the media query for checking whether the user is on a small device and simply added it to an @import syntax used to load the tablet.css file.

Similarly to using separate style sheets linked to the page in your <head> tag, style sheets loaded via @import increase the number of HTTP requests. However, in addition to this, the @import can prevent style sheets from being downloaded concurrently.

Using Media Queries in CSS

Rather than separate the CSS into separate style sheets, you can define the media queries within the site's style sheets. This allows you to define new styles and apply overrides to existing styles for when the condition is true, taking advantage of the cascading nature of CSS.

To write a media query in the CSS document, you can use the @media syntax. A breakdown of how to use this syntax is:

1. The @media declaration
2. One or more media expressions

An example of the @media syntax would therefore be:

```
@media only screen and (min-width: 768px) and (max-width: 1023px){ /* Your styles. */

}
```

Using Fluid Layouts

Types of Layouts

When it comes to the structural layout of the page, there are multiple types of layouts from which you can choose. The three most popular types of layouts are fixed width layout, fluid layout, and elastic layout, each having its own benefits and disadvantages.

When deciding upon which type of layout you want to use, it is important to consider the user's experience. For a responsive site, your layout choice needs to work well across a significant number of devices. With the sheer number in use, it is important to choose a layout type that allows you to provide the best experience to the most number of users. You also need to take into consideration the very possible chance that your site could be accessed on anything ranging from the smallest of smartphones to an 85-inch television in a living room.

Let's take a look at the different layout styles and consider the benefits and disadvantages of each, with the intention to learn how to pick the best bits from each in order to build a great experience for web site users.

Fixed Width Layouts

As the name suggests, fixed width layouts are built primarily with a wrapper that has a fixed width. This is then positioned on the screen, typically being centered in the browser viewport. Regardless of the size of screen being used to access the site, the fixed width layout will maintain its fixed width.

Historically, when developers built web sites, they were building from a design, which has a fixed width. Typically, designers would design sites to be 960px wide, as this was the ideal width for using grid layouts because this number is divisible by 3, 4, 5, 6, 8, 10, 12, and 15. When developers proceeded to build these designs, they would then build the site at 960px wide, striving to be as pixel perfect with the build as possible. The idea here was that they could make the design look as fantastic in the browser as what had been presented to the client as flat designs. An example of a typical three-column fixed layout, with two sidebars and a main content area, is shown in Figure 4-1.

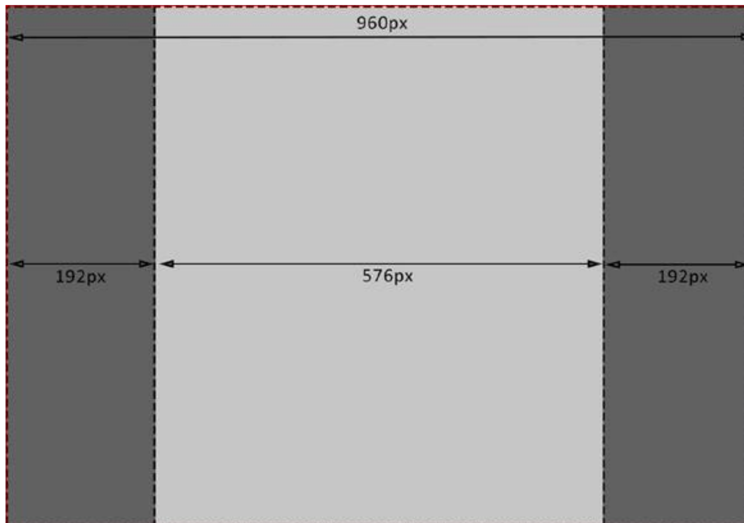

Figure 4-1. *An example of a 960px wide, three-column fixed layout*

If you look at how a typical fixed width layout is made up, you would normally have a wrapper with a fixed width of 960px, and inside of this you would then have your web site content, usually having the width defined in pixels. In the example in Figure 4-1, there is a main content area with a width of 576px with two 192px columns on either side.

Fixed width builds are still incredibly popular, and when you factor in the ability to create content including images that perfectly fit the design, it is easy to understand why. A current example of a fixed width web site for Samsung is presented in Figure 4-2.

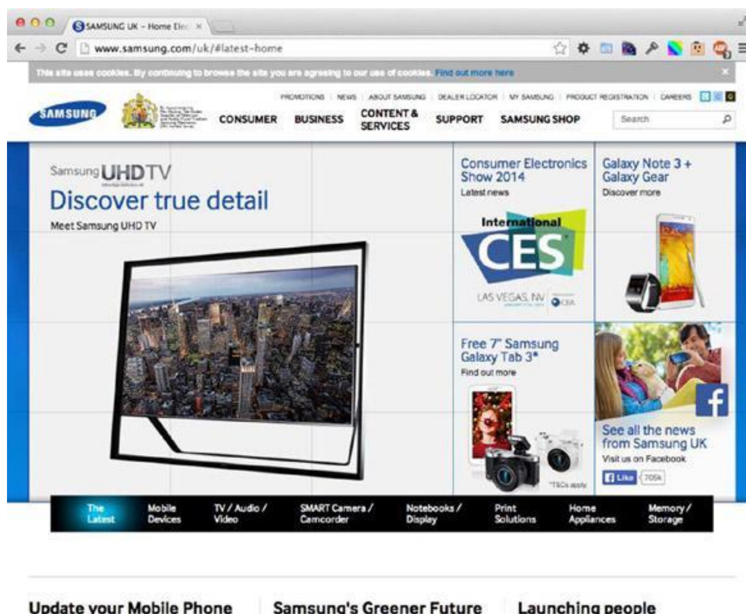

Figure 4-2. *The Samsung web site is a good example of a fixed layout*

The Samsung web site is built to be 960px wide, enabling the site to target a wide variety of popular viewports. Although not responsive, the site is still usable on a mobile device; it initially loads full screen with the full site visible, then, to interact with the site, the user is able to take advantage of the browser built-in user interactions of pinch, pull, and double-tap to zoom and navigate the content. Although this is usable, it doesn't provide the fantastic user experience that we have come to expect on our mobile devices.

If you take a look at the Samsung site in Mobile Safari on an iPhone, you would see the image shown in Figure 4-3.

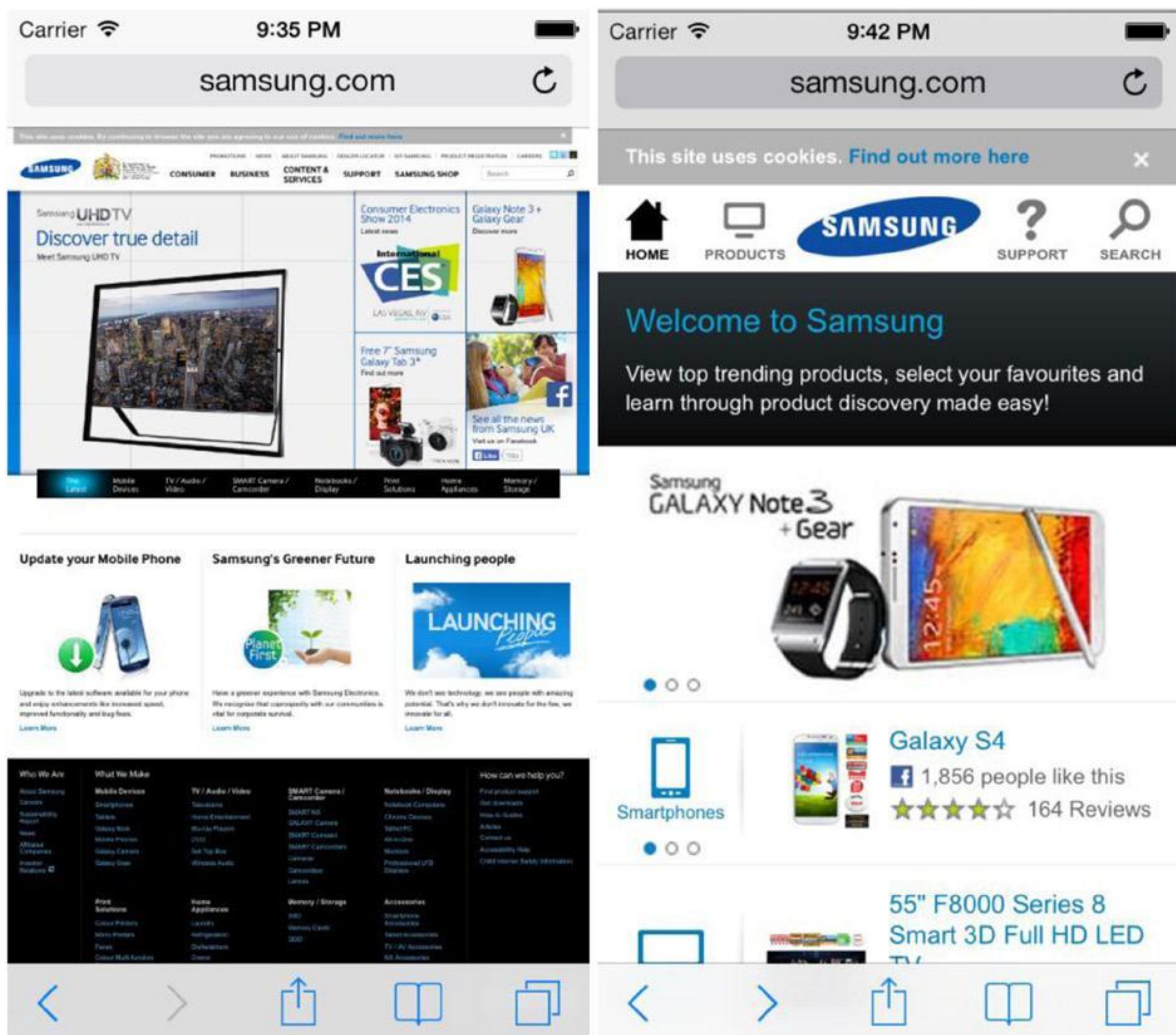

Figure 4-3. *Samsung desktop site (left) vs. Samsung mobile site (right)*

Samsung does provide a specific mobile site, which is shown in Figure 4-3, however, it does not share the same content as the desktop site. This means that users could still end up using the desktop site on their mobile devices to access content that is unavailable otherwise.

One of the main issues of fixed width design is that you are making a judgment call regarding what width your sites should be designed and built to. The limitation of making this kind of decision is that there is a huge range of browser widths to support. The result of this is that on a browser window smaller than the chosen fixed width you get horizontal scrolling, and on larger browser windows you have excess space to the sides of your site that could be better utilized.

When Apple launched the iPhone in 2007, the majority of web sites were fixed width so developers tried to compensate for this by making sites by default start zoomed out (as per Samsung), meaning that users would be able to see the full width of the site despite it being too big for their devices. Although this offered a better experience than only getting to see a corner of a site, it still could be quite frustrating to navigate a large site.

Using a fixed width design isn't necessarily a bad decision, and it is important to consider what your web site will be used for and its target audience. A fixed width site makes sense for sites where you want to retain the same layout and proportions across all viewports.

Elastic Layouts

Elastic layouts do not define widths in pixels but instead are measured in ems. The em unit is a multiple of the font size, so if you set your font size to 16px, then a width of 2em would be equal to 32px. This means that the if the user changes the font size while viewing the site, the layout of the site will also change proportionally to the increase or decrease in font size.

Elastic layouts give the developer more control because the design proportions stay intact when the user resizes the text in the browser. What this means is that if a user needs to increase the font size used on your site, the experience they receive on your site is no different from that of a user viewing the site at the original font size. The elements of the site resize in proportion to the font size. This means that elastic layouts work really well at enabling developers to ensure that their sites are accessible to all users.

If you take the original fixed width web site design, using a base font size of 16px, you can easily convert the layout to use em instead of pixels by dividing each of the pixel column widths by the font size, in this case 16. Figure 4-4 shows the same layout used in Figure 4-1 for the fixed layout example converted to an elastic layout.

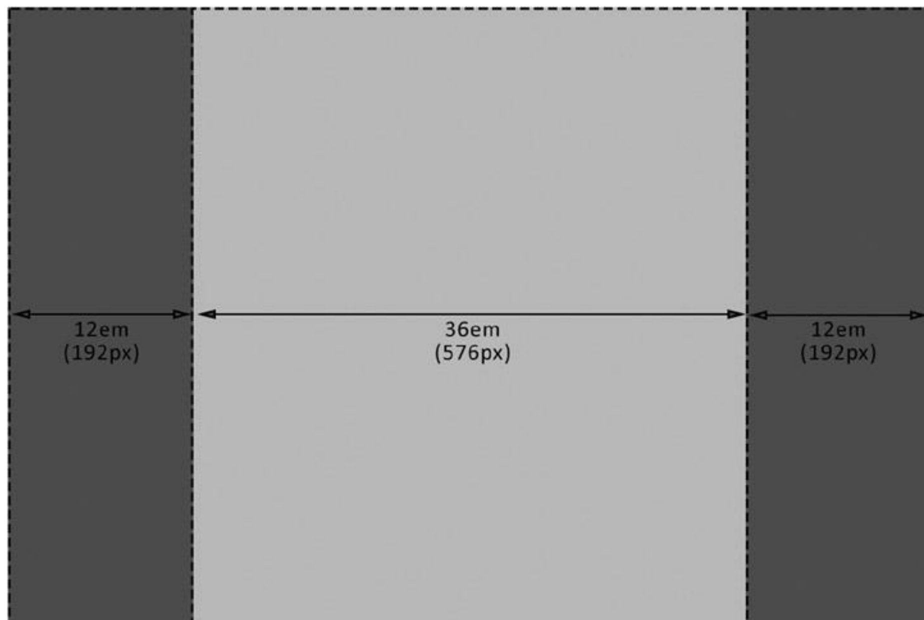

Figure 4-4. *An example of a 960px wide, three-column elastic layout with em conversion*

The example now has the column width defined in ems rather than pixels. The calculation from pixels to ems is very easy to perform:

Width in pixels / base font size = Width in em

Using this simple formula, you can easily calculate the widths of your elements in ems. In Figure 4-4 you can see how to calculate the element widths by substituting the pixel widths into the calculation:

576 / 16 = 36em

192 / 16 = 12em

It is important to be aware that ems can have a decimal value, so if your calculation doesn't result in a round number, it isn't a problem.

An example of a site that uses elastic layouts is the Northern Ireland Community Archives microsite about the Plantation of Ulster, as shown in Figure 4-5.1

Figure 4-5. The Plantation site is a good example of an elastic layout

This web site uses ems to define the widths of both the outer wrapper and the columns of its layout. If the base font size is increased, the whole site then resizes to accommodate the larger text, as shown in Figure 4-6.

Figure 4-6. *The Plantation site after increasing the base font size*

One of the initial problems with elastic layouts is that because the width of the site is based on the base font size, if the user increases the font size beyond a certain point, the site will become very large in the viewport, potentially causing the browser to have horizontal scrollbars.

Another problem with elastic layouts is that they are essentially still a form of fixed width layout, by which I mean that the elasticity is based on the base font size, so unless the font size is changed by the user, the site will stay the base width the developer intended. What this means is that while elastic layouts offer a more accessible approach than

a pixel-based fixed layout, the layout still appears to snap into position at each of the breakpoints because the CSS inside each of the media queries is applied; thus, there is no fluidity between them as the site adapts to different sizes.

One of the reasons that elastic layouts are not more popular is because of the difficulty that is posed by having to calculate the widths in ems. It isn't immediately clear what a width of 36em would be in pixels until you look at the base font size. With a base font size of 16px, the width would be 576px; however, by simply changing the base font size to 14px, the width would be reduced to 504px. Having to perform this (albeit, simple) calculation adds an extra step for the developer.

Another issue you might face is that ems are calculated relative to the parent; therefore, if the parent defines a different font size than the body, you might find the width of the element is not what you would expect. This is because you have likely calculated the width based on the font size of the body; to fix this you simply redo the calculation based on the font size of the parent element. To avoid this issue completely, one option is to use rem rather than em for the font size, which means all of the font sizes will be relative to the base font size regardless of the font size of the parent's elements, the caveat being that rem does not work in legacy browsers such as Internet Explorer 8 and earlier.

Fluid Layouts

Fluid layouts, also known as liquid layouts or relative layouts, change in width dependent on the user's viewport. Unlike fixed layouts, where widths are defined in pixels, you instead define the widths as percentages, where the percentage is referencing its portion of the viewport, as shown in Figure 4-7.

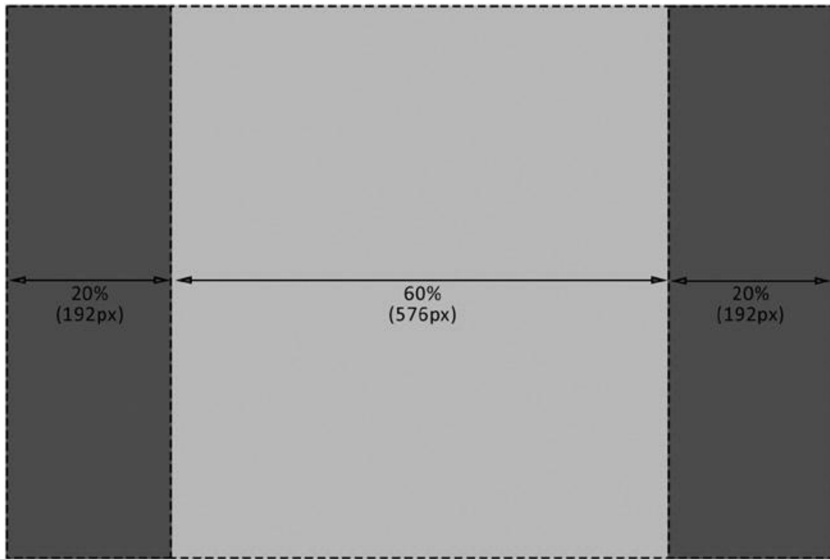

Figure 4-7. *An example of a fluid web site, with browser currently open at 960px wide*

If you take the example in Figure 4-1 of the fixed width layout and turn the dimensions into percentages, you would have two columns with a width of 20 percent and a single column with a width of 60 percent. With a browser viewport at 960px, the equivalent pixel values are the same as those in the fixed example (pixels shown in brackets), as shown in Figure 4-8.

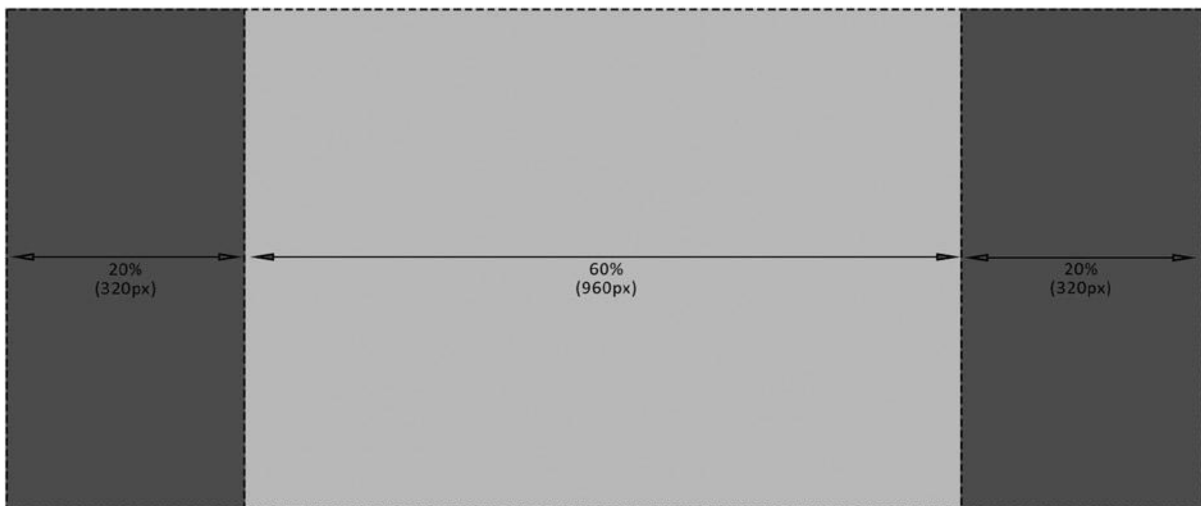

Figure 4-8. *An example of a fluid web site, with the browser open at 1,600px wide*

If you were then to resize the browser to have a viewport width of 1,600px, the web site would simply naturally scale up to the full width of the browser. The smaller columns would now have a width of 320px and the larger column a width of 960px. The benefit of this is that the content naturally fills the page, potentially scaling up any images and flowing the text to fill the available space.

Yahoo! recently redesigned the Flickr site using a fluid layout approach (as shown in Figure 4-9), taking advantage of the additional space available to allow them to increase the size of the images, and depending on the screen size, increase the number of images per row. The benefit here is that users of the site could browse through more images quicker and be more engaged with the site.

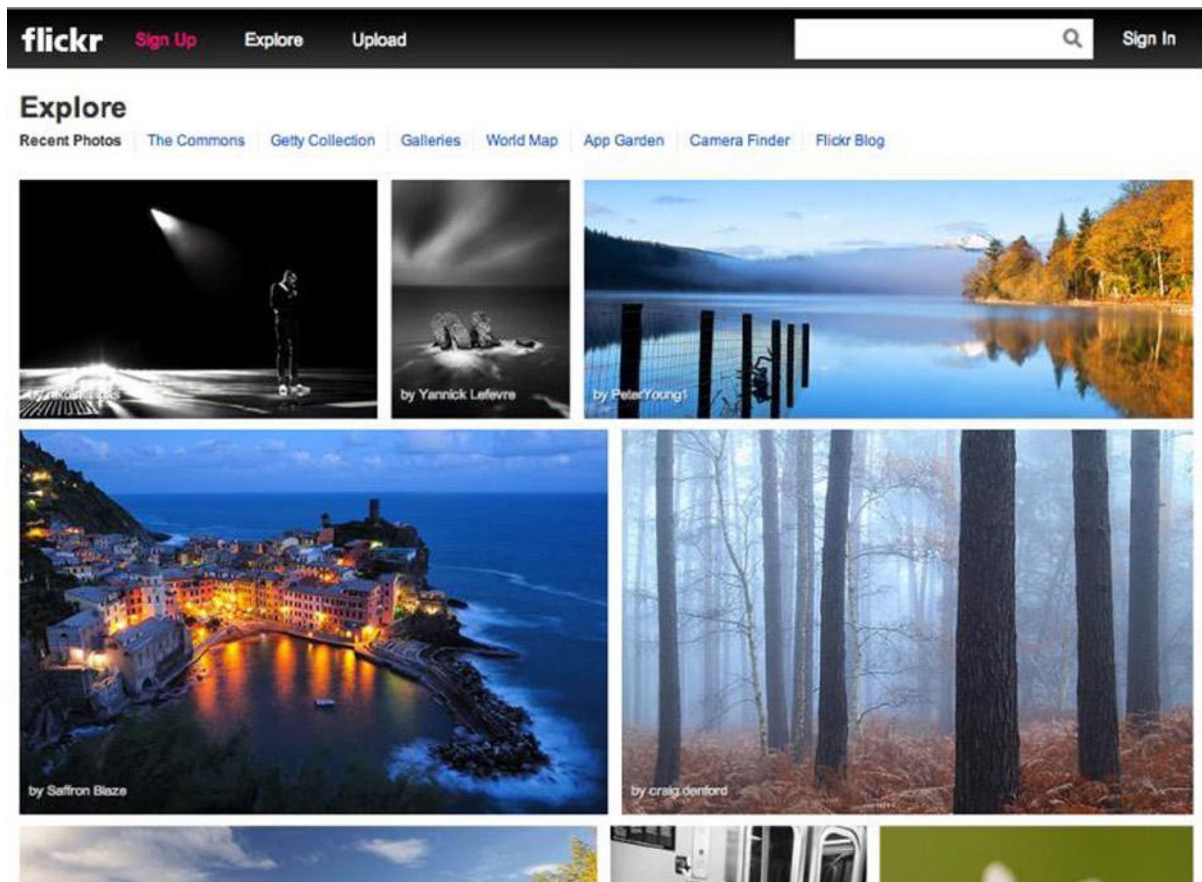

Figure 4-9. *The Flickr web site is a good example of a fluid web site*

If you were to resize the browser window, the Flickr site simply scales with the browser, making better use of the available space. Images either increase in size or allow for more in a row, utilizing the users' viewing space and maximizing the amount of material they are able to view at any given time. You can see this wider view of the Flickr site in Figure 4-10.

Figure 4-10. *The Flickr web site when viewed on a wider screen*

The benefit to Flickr of using a fluid layout is clear; the imagery can fill up all available space, allowing the user to properly appreciate the photos. The improvement of the experience to the users can also be measured by the increase in engagement the site has received. Although Yahoo! has not released any firm figures, a Flickr user named Thomas Hawks[2] did some research on his own. Within six days of the redesign launching, 80 million new photos had been uploaded to the site, in comparison to the six days prior to the redesign, when users had uploaded only 47 million images.

A fluid layout is not only beneficial to sites like Flickr, which is solely imagery, but it also of benefit to sites where content is tiled. An example is Pinterest, which, although also quite heavy on images, has other content displayed. For Pinterest, users will often skim through the boards that interest them, looking for things that grab their attention, and by using a fluid layout, Pinterest is able to show the maximum amount of content the user is able to view at that time, allowing them to find what they want, faster. The fluid width, tiled interface of Pinterest is shown in Figure 4-11.

Figure 4-11. *The tiled interface of Pinterest benefits from fluid design to show more content*

One of the benefits of using a fluid layout is that it can be a lot more user friendly simply because of the way it adjusts to the user's device. Rather than having excess white space, a fluid layout can take advantage of the space available, allowing content to fill all the available space, and on larger devices, it allows the content be more spaced out with extra breathing room.

Although fluid layouts are incredibly powerful, they do have some disadvantages, the most important of which is that they can result in issues in the way the content is rendered. Because a fluid layout could be viewed at any browser width, you may find that you face issues with how your images look. Although it is possible to scale your images as the browser resizes, it is important to be aware that some images will not scale well. Images containing focal points will suffer, as scaling them down will not only lose the impact, but also make the focus too

small. Similarly, a low-resolution picture, when scaled up, could significantly degrade in quality.

Aside from issues with content rendering, a fluid width site may face issues with some of its interactive functionality. It is common for web sites to use JavaScript to augment the user's experience, for example, to display equal height columns. The problem comes with when the user resizes the browser; the text reflows and the columns then become too tall or too short. Therefore, the JavaScript needs to be adapted to recalculate the heights upon resizing.

Why Use Fluid Layouts for Responsive Design

Having explored fixed, fluid, and elastic layouts, it is important to get a good understanding of the benefits and disadvantages of each approach. I have not, however, really discussed how this all relates to responsive design.

In Chapter 3, I explained how you can use media queries to adapt your web sites to different viewports. The problem with doing this is that with both fixed and elastic layouts, they snap between two or more different layouts that are adapted to different screen sizes. Rather than being responsive to all sorts of different devices, they are instead adapted to work across a few different common screen sizes.

But devices no longer conform to a few different common screen sizes; instead developers now face browsers with viewports of any dimension, viewed on devices ranging from an 85-inch television to a 3.5-inch mobile phone screen. Developers must have the mindset of wanting their web sites to look great regardless, not just on a few choice screen sizes. If the user were to resize their browsers, you would want the resize to be smooth and flowing, with the layout simply working at any given resolution.

When considering your choice of layout style, you should ensure that you are familiar with the alternative options available, which is why I have covered the advantages and disadvantages in the previous sections. When building web sites, you should feel comfortable with your choice of layout style and adapting them to suit your needs, not be restricted by them. That being said, responsive design really requires a fluid layout to deliver the best user experience across the widest range of devices with minimal interruptions between different viewport sizes. This doesn't mean you can't have breakpoints, which causes the layout to snap to change it to better take advantage of the available screen space. This is, in fact, encouraged;

you will want to be able to take advantage of the space you have available the best way you can.

Principles When Working with a Fluid Design

Building a fluid design can bring a wide variety of benefits to responsive design; however, there are some important principles to bear in mind to ensure your web site remains usable regardless of the size of the viewport.

The key principles you should try to follow are:

1. Do not use fixed heights.
2. Do not necessitate horizontal scroll bars.
3. Think about how your images look at different sizes.
4. Think about wrapping content.
5. Think about spacing.
6. Think about the length of your lines of text.

Let's explore these in more detail, looking at potential ways around the problems you might encounter.

Do Not Use Fixed Heights

If you are used to building fixed layouts, you most likely have encountered the problem where you have defined a fixed height for an element, but when the content was changed, the height either had excess spacing or the content overflows outside the defined height.

With the variable widths that come with fluid design, this issue is a lot more common, as the way in which the content wraps will change based on the size of the viewport, so setting a fixed height in the CSS becomes impractical.

Unfortunately, sometimes it is integral to the design to show content in equal columns, each with its own background color. This can be tricky to achieve without defining the height in the CSS; so the options are either to create faux columns in the CSS or use JavaScript to dynamically set the height of the columns.

Using CSS to Create Faux Columns

It is possible to create the effect of columns simply by using some CSS.

For a two-column layout, where you know that one column will always be taller than another, simply set the parent element's background color to that of the shorter column. Then by setting the background color of longer column's element, you have created the effect of having two columns. Let's take a quick look at the code to handle this, starting with some simple HTML:

```
<div class="col-
container">
<aside class="col">
        Sidebar
    </aside>
    <div class="col main">
        Main Content Area
    </div>
</div>
```

As mentioned, you will apply the background for the columns to the column container and to the main content column. To achieve the column effect, you would also defined the columns as a width of 50 percent, floated left so they sit next to one another:

```
.col-container{
    background: #000;
    color: #fff;
}
.col-container:after{
    content: ' ';
    clear: both;
    display: block;
}
```

```
.col{
	float: left;
	width: 50%;
}
.col.main{
	background: #999;
}
```

It is important to note that you have added a pseudo-element to the column container to clear the floated columns. This is necessary so that the browser calculates the height of the column container itself. If your site is already using the clearfix class for this, you could choose to add this to your column container rather than add a pseudo-element specific to that element.

If you check this out in your browser, the layout has the two columns with equal height as expected, as shown in Figure 4-12.

Figure 4-12. Two columns with equal height using CSS

Although this is pretty simple, this method for two columns works across all browsers and is very effective, with the caveat being that this approach only works well for two columns and it requires that you know which is the taller column.

In the case where you have more than two columns, you could choose to use a CSS3 gradient on the parent to achieve the same effect. However, bear in mind this approach will only work in Internet Explorer 10 or later. Extending the example from above, you would add an additional column to the HTML:

```
<div class="col-
container">
<aside class="col">
        Sidebar
    </aside>
    <div class="col main">
        Main content area with our main body content
    </div>
    <div class="col">
        Related content
    </aside>
</div>
```

You can then simply adapt the existing CSS as you no longer need to add a background to the main column. Therefore, you can remove that part of the CSS. You will then add the CSS3 background gradient to the column container. In the example that follows, you only include the W3C CSS3 property, and it is important for live code to add the prefixed versions to ensure browser support:

```
.col-container{
```

```
	background: linear-gradient(to right, #000000 0%,#000000 33%,#a0a0a0 33%,#a0a0a0 66%,#a0a0a0 66%,#707070 66%);
	color: #fff;
}
.col-container:after{
	content: ' ';
	clear: both;
	display: block;
}
.col{
	float: left;
	width: 33.3%;
}
```

When looking at the results of running this in the browser, the new column appears with the correct background color, as you can see in Figure 4-13.

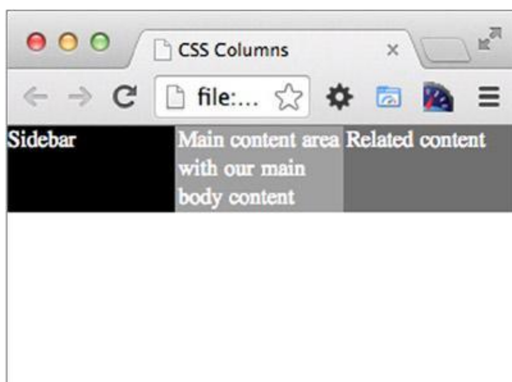

Figure 4-13. *Three columns with equal height using CSS*

If you need support for older versions of Internet Explorer or you want to support a dynamic number of columns, you should perhaps look at using JavaScript.

Using JavaScript

Having already looked at how to use CSS to create a faux columns effect, let's look at how to use JavaScript to create columns of equal height. The JavaScript approach does not try to fake the appearance of columns, but instead makes all the columns the same height.

The approach to using JavaScript to create equal height columns is to loop through each of the columns, find which is the tallest column, and then set all the columns to the height of the tallest.

To demonstrate this, let's look at a simple example, beginning with some HTML:

```
<div class="col-container">
    <aside class="col nav">
        Sidebar
    </aside>
    <div class="col main">
        Main content area with our main body content
    </div>
    <div class="col related">
        Related content
    </aside>
</div>
```

You then need to style the columns so they each have a different background color:

```
.col-container{
    color: #fff;
}
```

```
.col-container:after{
	content: ' ';
	clear: both;
	display: block;
}
.col{
	float: left;
	width: 33.3%;
}
.col.nav{
	background: #aaa;
}
.col.main{
	background: #000;
}
.col.related{
	background: #999;
}
```

With this in place, the columns now all sit nicely, each with a unique background color; however, the heights of the columns are still unequal. You can see how this looks in Figure 4-14.

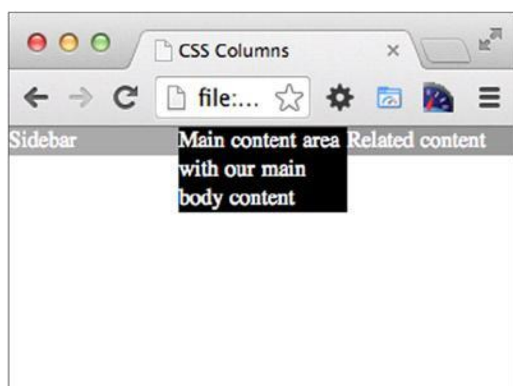

Figure 4-14. Three columns with unequal height before adding the JavaScript

To progress to making the columns equal, you will need to add some JavaScript. In the below example I have used document.getElementsByClassName, which is only supported by Internet Explorer 9 or later. However, you can use a polyfill to enable support for this method, one such polyfill is available in this Gist code: https://gist.github. com/eikes/2299607.

```
<script>
    var columns =
        document.getElementsByClassName('col')
        , height = 0;
    //Loop through columns and find the tallest
    columns for (var i = 0; i < columns.length; i++) {
        if(height < columns[i].clientHeight){
            height = columns[i].clientHeight;
        }
    }
    //Apply the max height to all columns
    for (var i = 0; i < columns.length; i++) {
        columns[i].style.height = height + "px";
    }
</script>
```

The equal column JavaScript simply iterates through all of the columns to find the tallest element. It then iterates through the elements again to define the height. You can see the JavaScript working in Figure 4-15.

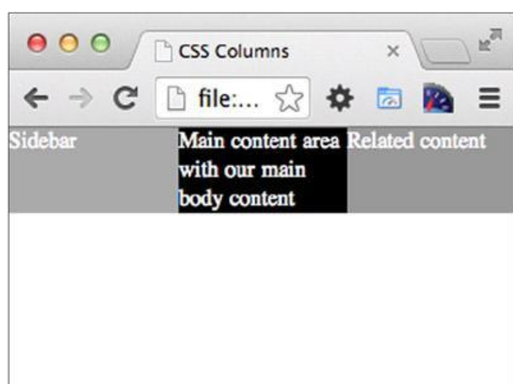

Figure 4-15. *Three columns with equal height after adding the JavaScript*

With a fixed, non-responsive design, simply running this JavaScript when the page has loaded is enough. However, with a fluid, responsive design, the user may resize the browser so the JavaScript then needs to be able to update the column height on resize:

```
<script>
    var equalColumns = function(){
        var columns = document.getElementsByClassName('col'), height = 0;
```