Media Queries

Knowing how to develop responsive websites has become an indispensable skill for today's web developer.

What makes a website responsive then? How should we code? These are important questions that we will cover in this chapter.

You'll learn:

- What is required to make responsive websites
- How to code responsive websites, mobile-first

Things Required For a Responsive Website

There are only two things you will need to make a website responsive.

1. Meta Viewport Tag
2. Media Queries

Meta Viewport Tag

Hand-held devices today usually have a screen width that is larger than the actual width of the device. iPhone 5s for example, has a device resolution of 640px x 1136px. The actual screen size however, is half of that, at 320px x 568px.

Browsers render the width according to screen resolution by default. They will think our iPhone 5s has a width of 640px instead of 320px. This would mean that everything will be zoomed out to be half its intended size on an iPhone 5s if we don't do something about it.

To combat this, we need to add a meta viewport tag to the `<head>` of the website. This will tell the browser to render the width of the page as the width of its own screen.

```
<head>
  <meta name="viewport" content="width=device-width, initial-
scale=1">
</head>
```

Now, the iPhone 5s will render at 320px where we intend it to.

Media Queries

If you resize your browser on a responsive website, you will notice that the layout of the website changes at certain points. These are called breakpoints.

Media queries are used to specify where these breakpoints are, and how the browser should respond when the conditions are matched. They have the following syntax:

```
/* CSS */
@media [<media-type> and] (<condition>) {
  /* properties here */
}
```

`@media` is the keyword for media queries. It signifies the start of a media query.

`<media-type>` is an optional argument that tells browsers to filter for a specific media type. The most common media types are `all`, `screen` and `print`. The `print` media type only applies when the page is printed out while `screen` works when the site is viewed on the screen. `<media-type>` defaults to all when not specified, will apply to both `print` and `screen`.

`condition` is a required argument in a media query that allows us to specify how the browser should render a page if the condition is true. They are written in a property and value pair like:

```
@media (min-width: 600px) {
  /* This applies when the browser width is equal to or larger
than 600px */
}
```

You can chain more than one conditions with the `and` keyword, like:

```
@media (min-width:600px) and (max-width: 1200px) {
  /* This applies only when the browser width is BETWEEN 600px
and 1200px */
}
```

Note: There are more conditions that you can potentially use with media queries. We will only cover `min-width` and `max-width` since they will be used the most.

Media queries have to be written as the first level wrapper in a CSS file. As such, every media query declaration would have the following format:

```
/* CSS */
@media (min-width:600px) {
  .content {
    /* Content properties */
  }
}
```

Writing this way could become problematic when we have hundreds of selectors that use a specific media query. People used to throw in these media queries together at the end of the css file:

```
/* Standard CSS properties here */

@media (min-width: 600px) {
  .selector1 {}
  .selector2 {}
  .selector3 {} /* ... */
}

@media (min-width: 1200px) {
  .selector1 {}
  .selector2 {}
  .selector3 {} /* ... */
}
```

This way of coding decouples the media queries from the selectors themselves, and makes it hard for us to understand the code when we look at it at a later time.

There are only 3 selectors above, imagine if you have over 200 selectors in each media… How much of a nightmare would that be?

Thankfully, as of Sass v3.2, media queries can be placed within the selectors themselves:

```
// Scss
.content {
  @media (min-width:600px) {
    @include span(3 of 4);
  }

  @media (min-width:1200px) {
    @include span(8 of 12);
  }
}
```

This way of coding couples the styles from the same selector at different breakpoints together, allowing you to easily understand what is going on.

Sass automatically translates it into the CSS code format when compiled:

```
/* CSS */
@media (min-width: 600px) {
  .content {
    /* CSS properties at 600px */
  }
}
@media (min-width: 1200px) {
  .content {
    /* CSS properties at 1200px */
  }
}
```

You may be concerned about performance since we're introducing multiple media queries when we code this way. You don't have you worry.

This is because the difference in rendering time between a stylesheet with 40 media queries and another with 2000 media queries is only a mere 100 milliseconds difference in a test done by Aaron Jenson.

Since we now know how to work with media queries, let's find out how to write them, mobile first.

Mobile First Media Queries

Mobile-first and desktop-first media queries have subtle differences.

A mobile-first approach to styling means that styles are applied first to mobile devices. Advanced styles and other overrides for larger screens are then added into the stylesheet via media queries.

This approach uses `min-width` media queries.

Here's a quick example:

```css
// This applies from 0px to 600px
body {
  background: red;
}

// This applies from 600px onwards
@media (min-width: 600px) {
  body {
    background: green;
  }
}
```

In the example above, `<body>` will have a red background below 600px. Its background changes to green at 600px and beyond.

On the flipside, a desktop-first approach to styling means that styles are applied first to desktop devices. Advanced styles and overrides for smaller screens are then added into the stylesheet via media queries.

This approach uses `max-width` media queries.

Here's a quick example:

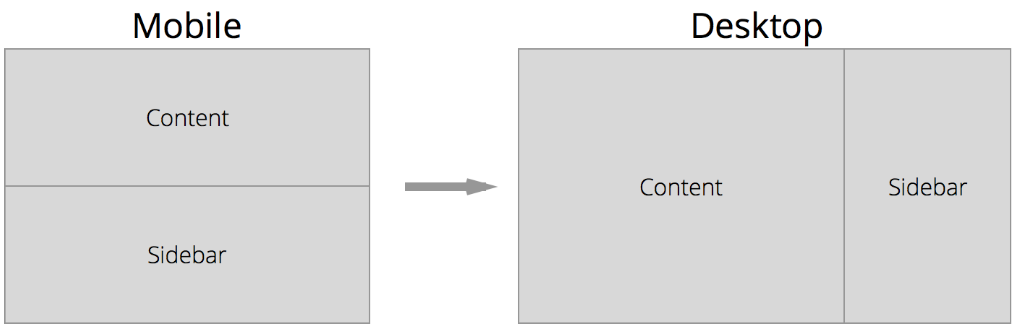
```css
// This applies from 600px onwards
body {
  background: green;
}

// This applies from 0px to 600px
@media (max-width: 600px) {
  body {
    background: red;
  }
}
```

`<body>` will have a background colour of green for all widths. If the screen goes below 600px, the background colour becomes red instead.

Why Code Mobile-first?

Code for larger screens is usually more complicated than the codes for smaller screens. This is why coding mobile first helps simplify your code.

Consider a situation where you have a content-sidebar layout for a website. `.content` takes up a 100% width on mobile, and 66% on the desktop.



Most of the time, we can rely on default properties to style content for smaller screens. In this case, a `<div>` has a width of 100% by default.

If we work with the mobile-first approach, the Sass code will be:

```
.content {
  // Properties for smaller screens.
  // Nothing is required here because we can use the default
styles

  // Properties for larger screens
  @media (min-width: 800px) {
    float: left;
    width: 60%;
  }
}
```

If we go with the desktop-first approach instead, we will have to restore the default properties for smaller viewports most of the time. The Sass code for the same result is:

```
.content {
  // Properties for larger screens.
  float: left;
  width: 60%;

  // Properties for smaller screens.
  // Note that we have to write two default properties to make
the layout work
  @media (max-width: 800px) {
    float: none;
    width: 100%;
  }
}
```

With this one example, we save two lines of code and a few seconds of mind-bending CSS. Imagine how much time and effort this will save you if you worked on a larger site.

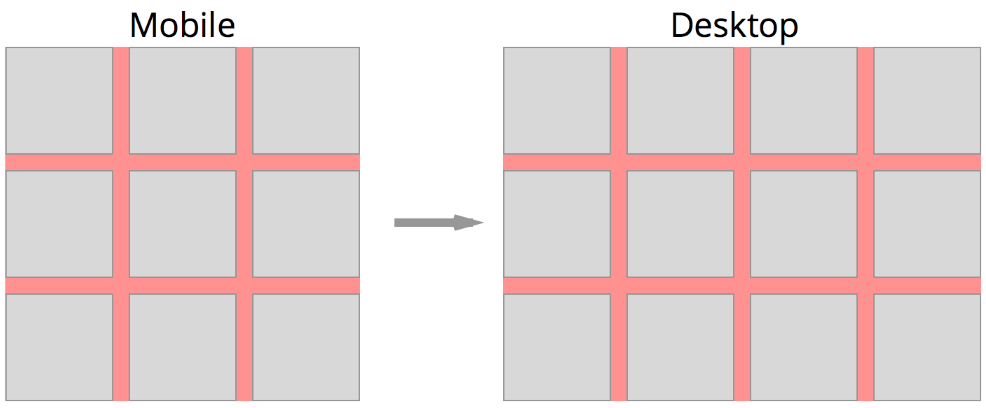
Most of the time `min-width` queries would be enough to help you code a website. There are however instances where a combination of both `min-width` and `max-width` queries helps to reduce complications that pure `min-width` queries cannot hope to achieve.

Let's explore some of these instances.

Using Max-width Queries With A Mobile-First Approach

`Max-width` queries come into play when you want styles to be constrained below a certain viewport size. A combination of both `min-width` and `max-width` media queries will help to constrain the styles between two different viewport sizes.

Consider a case of a gallery of thumbnails. This gallery has 3 thumbnails in a row on a smaller viewport and 4 items in a row on a larger viewport.



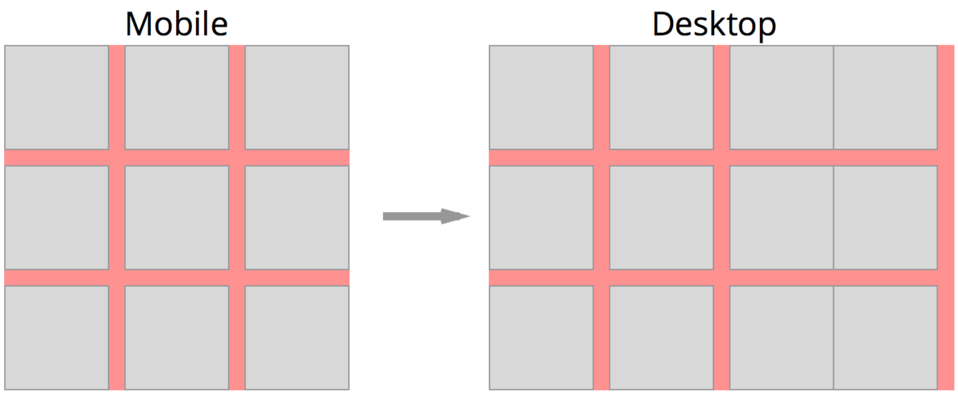
```
.gallery__item {
  @include span(4 of 12);
  margin-bottom: gutter(12);
}
```

We will also have to give the final (3rd item) on the row the `last` keyword to make sure it doesn’t get pushed down into the next column.

```
.gallery__item {
  @include span(4 of 12);
  margin-bottom: gutter(12);
  &:nth-child(3n) {
    // `last()` is a mixin that removes the right margin
    @include last;
  }
}
```

This code must also work for the case where there are four items in the row. If we go according to the min-width query we had above…

```scss
.gallery__item {
  @include span(4 of 12);
  margin-bottom: gutter(12);
  &:nth-child(3n) {
    @include last;
  }

  @media (min-width: 800px) {
    @include span(3 of 12);
    &:nth-child (4n) {
      @include last;
    }
  }
}
```



This doesn’t work properly because we specified that every 3rd item should have a `margin-right` of 0px. This property gets cascaded towards a larger viewport and breaks the pattern we wanted.

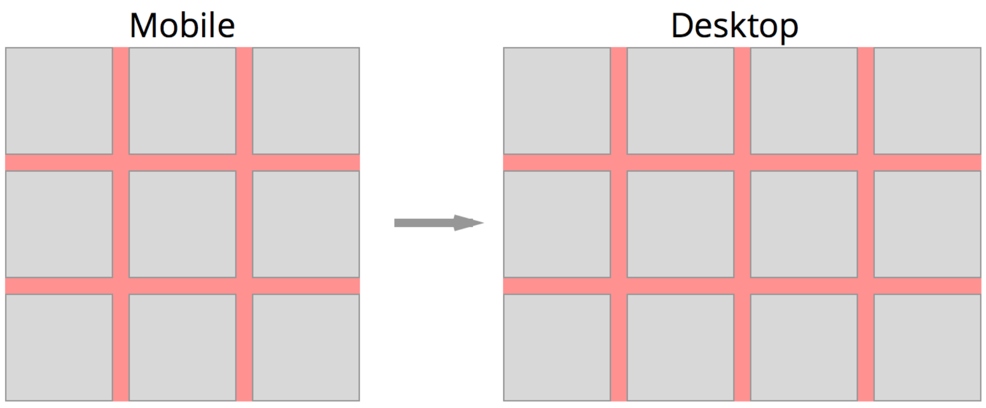
We can fix it by resetting the `margin-right` property of every 3rd item to the correct gutter:

```
.gallery__item {
  // ...
  @media (min-width: 800px) {
    // ...
    &:nth-child (3n) {
      margin-right: gutter(12);
    }
    &:nth-child(4n) {
      margin-right: 0%;
    }
  }
}
```

It’s not a very nice approach since we are repeatedly redeclaring and removing the `margin-right` property of several items with the `last()` mixin.

A better way is to constrain `nth-child(3n)` selector within its rightful viewport by using a `max-width` query.

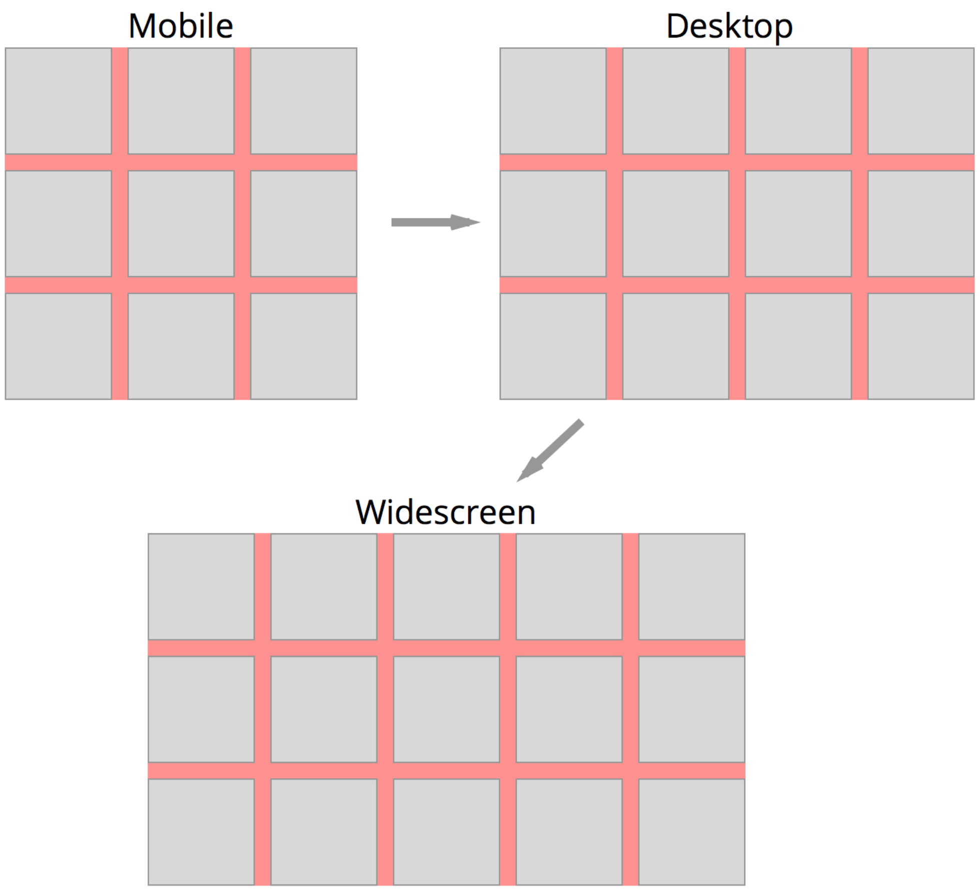
```
.gallery__item {
  margin-bottom: gutter(12);
  @media (max-width: 800px) {
    @include span(4 of 12);
    &:nth-child(3n) {
      @include last;
    }
  }

  @media (min-width: 800px) {
    @include span(3 of 12);
    &:nth-child (4n) {
      @include last;
    }
  }
}
```



This works because the `max-width` property limits the selectors to below 800px and the styles given within will not affect styles for any other viewports.

Now imagine if you have a larger viewport and you wanted to show 5 items per row in the gallery. This is when a combination of `min` and `max-width` queries come together.

```scss
.gallery__item {
  margin-bottom: gutter(12);
  @media (max-width: 800px) {
    @include span(4 of 12);
    &:nth-child(3n) {
      @include last;
    }
  }

  // Combining both min-width and max-width queries
  @media (min-width: 800px) and (max-width: 1200px){
    @include span(3 of 12);
    &:nth-child (4n) {
      @include last;
    }
  }

  @media (min-width: 1200px){
    @include span(3 of 15);
    &:nth-child(5n) {
      @include last;
    }
  }
}
```



That in the nutshell, is how to write mobile-first CSS.

A Quick Wrap Up

`Min-width` media queries are extremely helpful when it comes to coding responsive websites because it reduces code complexity. `Min-width` queries are, however, not the solution to every problem as you can see from the examples above. It can sometimes be beneficial to add `max-width` queries to help keep things cleaner.