## Responsive Overview

Responsive web design is the practice of building a website suitable to work on every device and every screen size, no matter how large or small, mobile or desktop. Responsive web design is focused around providing an intuitive and gratifying experience for everyone. Desktop computer and cell phone users alike all benefit from responsive websites.

### Responsive vs. Adaptive vs. Mobile

For some the term *responsive* may not be new, and others might be even more acquainted with the terms *adaptive* or *mobile*. Which may leave you wondering what exactly is the difference between all of them.

Responsive and adaptive web design are closely related, and often transposed as one in the same. Responsive generally means to react quickly and positively to any change, while adaptive means to be easily modified for a new purpose or situation, such as change. With responsive design websites continually and fluidly change based on different factors, such as viewport width, while adaptive websites are built to a group of preset factors. A combination of the two is ideal, providing the perfect formula for functional websites. Which term is used specifically doesn’t make a huge difference.

Mobile, on the other hand, generally means to build a separate website commonly on a new domain solely for mobile users. While this does occasionally have its place, it normally isn’t a great idea. Mobile websites can be extremely light but they do come with the dependencies of a new code base and browser sniffing, all of which can become an obstacle for both developers and users.

Currently the most popular technique lies within responsive web design, favoring design that dynamically adapts to different browser and device viewports, changing layout and content along the way. This solution has the benefits of being all three, responsive, adaptive, and mobile.

## Flexible Layouts

Responsive web design is broken down into three main components, including flexible layouts, media queries, and flexible media. The first part, flexible layouts, is the practice of building the layout of a website with a flexible grid, capable of dynamically resizing to any width. Flexible grids are built using relative length units, most commonly percentages or em units. These relative lengths are then used to declare common grid property values such as width, margin, or padding.

**Table of Contents**

* **Quick Start**
* **Using and Sharing Sass Variables**
* **Sass Files from 3rd Party Packages**
* **Conclusion and Demo**

## Quick Start

The steps to use Sass in Create React App are:

1. Install node-sass: npm install node-sass
2. Change .css files to .scss
3. Change any imports to use .scss

npm install node-sass -S

Once we've changed the file name from .css to .scss, we can import the Sass:

// replace

import "./styles.css";

// with

import "./styles.scss";

Done! Create React App will know to parse your .scss files and add the styles to your project.

## Using and Sharing Sass Variables

How do we share variables across files? We are able to import our Sass files from other Sass files. Let's say you create a variables file:

### variables.scss

$primaryColor: #BADA55;

We can import this inside of another file like we normally would in Sass:

### styles.scss

// import starting from the src/ folder

@import "variables.scss";

// can also be relative import

// @import "./variables.scss";

// we can use the $primaryColor variable now

h1, h2 {

color: $primaryColor;

}

## Sass Files from 3rd Party Packages

If we want to use any 3rd party libraries like Bulma or Bootstrap (Bulma is my favorite right now), we don't need to import the entire CSS library anymore.

With Sass in React, we can import just the files we need. First, we have to install Bulma.

npm install bulma -S

If we look at Bulma's GitHub in the sass/ folder, we can see where they place their .sassfiles. Notice they are using .sass and we are using the .scss variant. No problems, node-sass can read and @import both!

Import files from node\_modules using ~

The ~ let's webpack and Create React App know to look in the node\_modules/ folder for the files we need. Let's add a few of the files we need to our app:

### styles.scss

// import using ~

@import "~bulma/sass/utilities/\_all.sass";

@import "~bulma/sass/base/\_all.sass";

@import "~bulma/sass/elements/button.sass";

@import "~bulma/sass/layout/section.sass";

### App.js

function App() {

return (

<div className="App section">

<h1>Hello CodeSandbox</h1>

<h2>Start editing to see some magic happen!</h2>

<button className="button is-danger is-outlined">

Hello

</button>

</div>

);

}

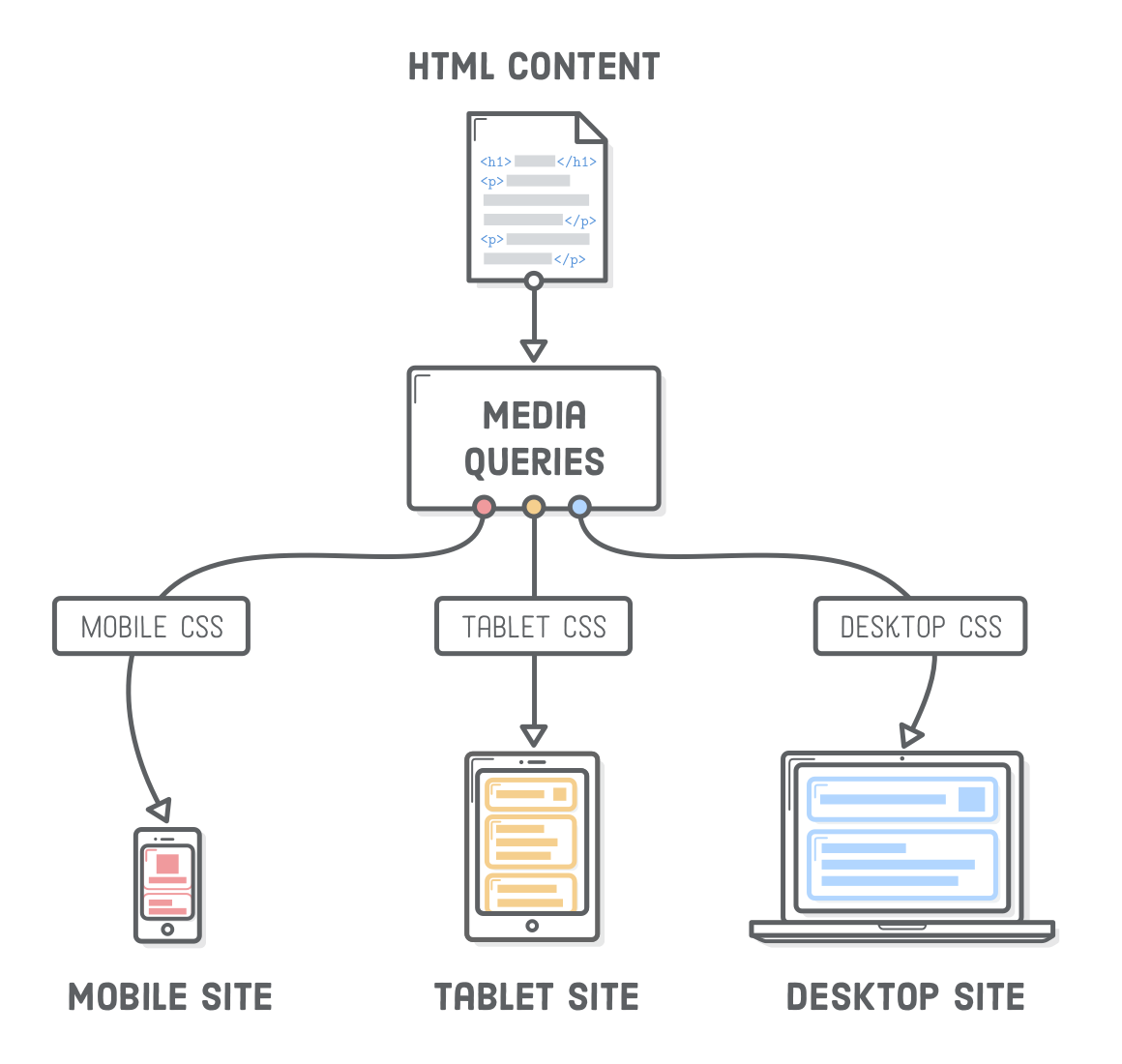
## Conclusion and Demo

Using Sass in React is a quick way to get styling in your app. It is also recommended to look at CSS-in-JS solutions so that we can create even more modular CSS in our component based React apps.

# Responsive design

“Responsive design” refers to the idea that your website should display equally well in everything from widescreen monitors to mobile phones. It’s an approach to web design and development that eliminates the distinction between the mobile-friendly version of your website and its desktop counterpart. With responsive design, they’re the same thing.

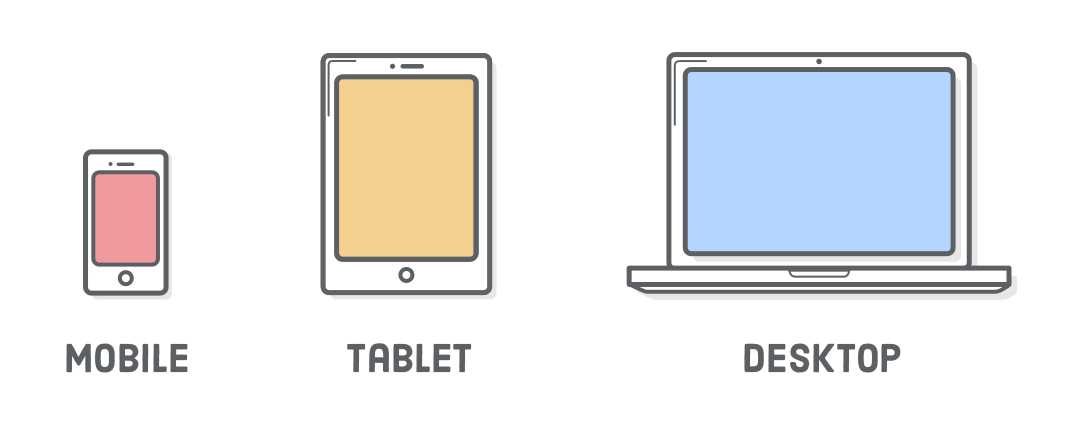
Responsive design is accomplished through CSS “media queries”. Think of media queries as a way to conditionally apply CSS rules. They tell the browser that it should ignore or apply certain rules depending on the user’s device.



Media queries let us present the same HTML content as distinct CSS layouts. So, instead of maintaining one website for smartphones and an entirely unrelated site for laptops/desktops, we can use the same HTML markup (and web server) for both of them. This means that whenever we add a new article or edit a typo in our HTML, those changes are automatically reflected in both mobile and widescreen layouts.

## css media queries

We’ll start small by simply updating the background color on the <body>element based on the device width. This is a good way to make sure our media queries are actually working before getting into complicated layouts.



Let’s differentiate between narrow, medium, and wide layouts by creating a new styles.css stylesheet and adding the following:

\* {

margin: 0;

padding: 0;

box-sizing: border-box;

}

/\* Mobile Styles \*/

@**media** only screen and (max-width: 400px) {

**body** {

background-color: #F09A9D; /\* Red \*/

}

}

/\* Tablet Styles \*/

@**media** only screen and (min-width: 401px) and (max-width: 960px) {

**body** {

background-color: #F5CF8E; /\* Yellow \*/

}

}

/\* Desktop Styles \*/

@**media** only screen and (min-width: 961px) {

**body** {

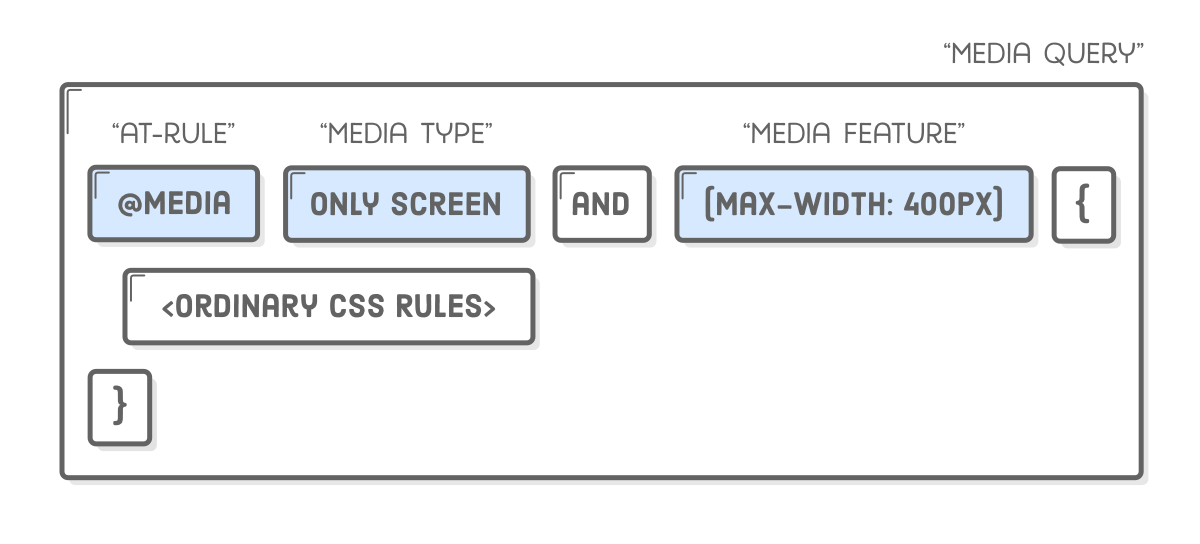
background-color: #B2D6FF; /\* Blue \*/

}

}

When you resize your browser, you should see three different background colors: blue when it’s greater than 960px wide, yellow when it’s between 401px and 960px, and red when it’s less than 400px.

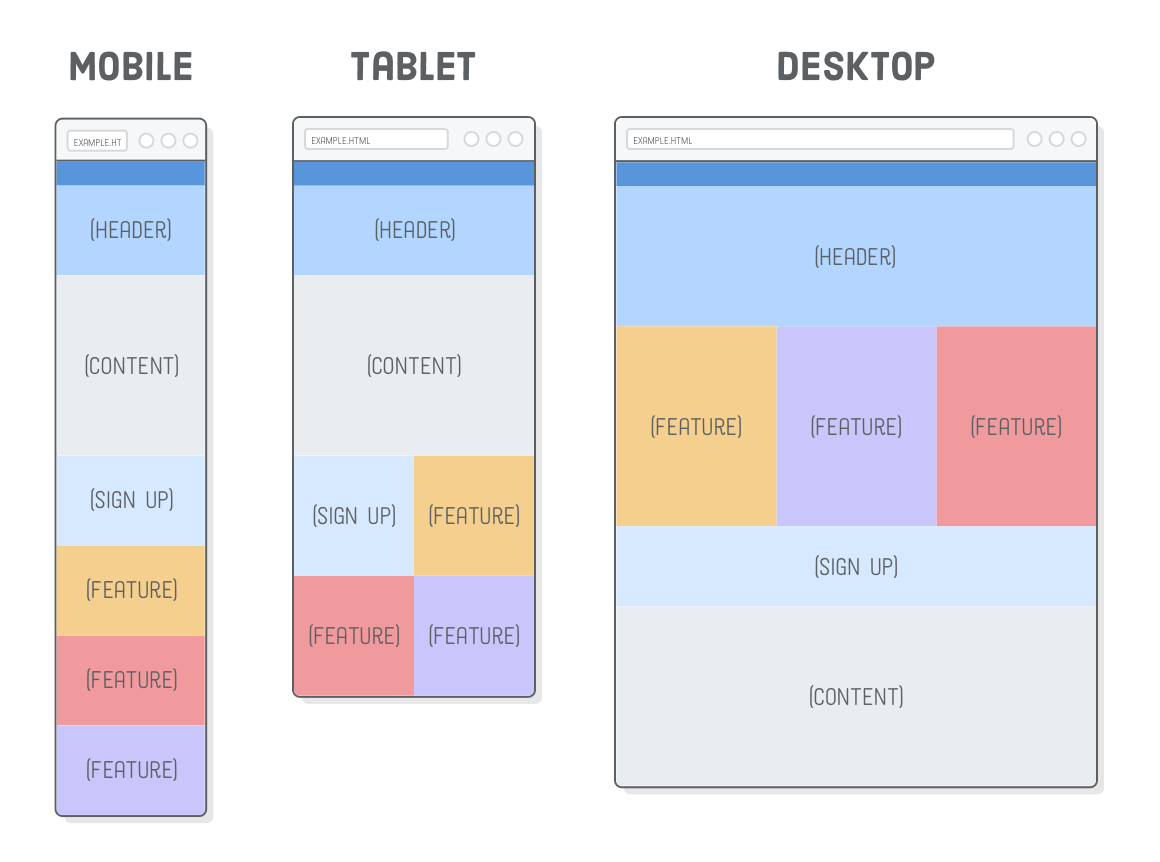
Media queries always begin with the @media “at-rule” followed by some kind of conditional statement, and then some curly braces. Inside the curly braces, you put a bunch of ordinary CSS rules. The browser only pays attention to those rules if the condition is met.



The only screen “media type” means that the contained styles should only be applied to devices with screens (opposed to printed documents, like when you hit **Cmd+P** in a browser). The min-width and max-width parts are called “media features”, and they specify the device dimensions you’re targeting.

## a few notes on design

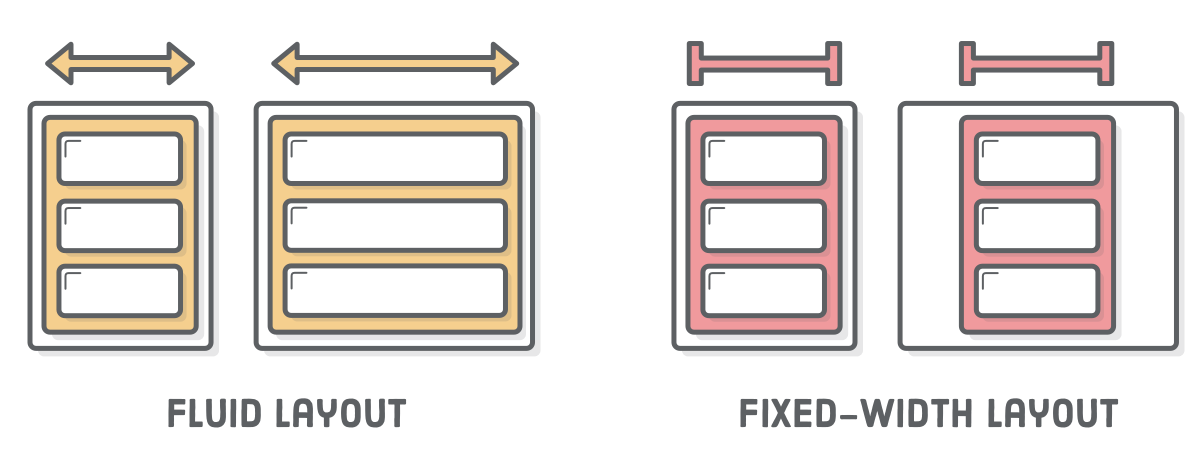
Ok, so @media is how we define different layouts for specific device widths, but what layouts are we actually trying to implement? The example web page for this chapter is going to look something like this:



In the real world, it’s up to your web designer to supply you with these kinds of mockups. Your job as a developer is to implement the individual layouts using media queries to separate out the various CSS rules that apply to each one.

There’s a few well defined patterns for how a desktop layout collapses into a mobile layout (we’re using “layout shifter”). A lot of these decisions are in the realm of design, which is outside the scope of this code-oriented tutorial; however, there are two concepts that you must understand as a developer:

* A “fluid” layout is one that stretches and shrinks to fill the width of the screen, just like the flexible boxes we covered a few chapters ago.
* A “fixed-width” layout is the opposite: it has the same width regardless of the screen dimensions (we created one of these in the CSS Selectorschapter).



In our example web page, the mobile and tablet versions are fluid, and the desktop version is fixed-width.

### choosing breakpoints

Most of those responsive design patterns have similar behavior, using fluid layouts for mobile/tablet devices and fixed-width layouts for wider screens. There’s a reason for this.

Fluid layouts let us target a range of screen widths instead of specific mobile devices. This is very important for web designers. When they set out to create a mobile layout, they aren’t trying to make something that looks good on an iPhone 6s, Galaxy S7, or iPad mini—they’re designing a fluid layout that looks good anywhere between 300 pixels and 500 pixels (or whatever).

In other words, the exact pixel values for the min-width and max-widthparameters in a media query (collectively known as the “breakpoints” for a responsive website) don’t actually matter. Our website doesn’t care about the specific device the user is on. All it needs to know is that it should display a layout that looks pretty at 400 pixels wide (or whatever).

## mobile-first development

Let’s dive right into implementing the above screenshots. It’s always a good idea to start with the mobile layout and work your way up to the desktop version. Desktop layouts are typically more complex than their mobile counterparts, and this “mobile-first” approach maximizes the amount of CSS that you can reuse across your layouts.

First, we need to fill in responsive.html’s <body> element with some empty boxes. Each box has an image in it so we can tell them apart a little bit easier.

<div class='page'>

<div class='section menu'></div>

<div class='section header'>

<img src='images/header.svg'/>

</div>

<div class='section content'>

<img src='images/content.svg'/>

</div>

<div class='section sign-up'>

<img src='images/sign-up.svg'/>

</div>

<div class='section feature-1'>

<img src='images/feature.svg'/>

</div>

<div class='section feature-2'>

<img src='images/feature.svg'/>

</div>

<div class='section feature-3'>

<img src='images/feature.svg'/>

</div>

</div>

And here’s our base styles, which should apply to all layouts (mobile, tablet, and desktop). Make sure to add these above the @media rules we created earlier and below the universal selector rule that resets our margins and padding:

.page {

display: flex;

flex-wrap: wrap;

}

.section {

width: 100%;

height: 300px;

display: flex;

justify-content: center;

align-items: center;

}

.menu {

background-color: #5995DA;

height: 80px;

}

.header {

background-color: #B2D6FF;

}

.content {

background-color: #EAEDF0;

height: 600px;

}

.sign-up {

background-color: #D6E9FE;

}

.feature-1 {

background-color: #F5CF8E;

}

.feature-2 {

background-color: #F09A9D;

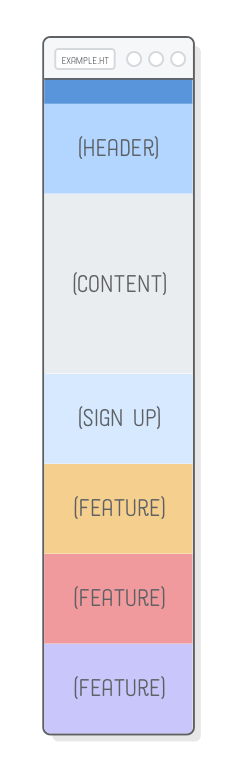
}

.feature-3 {

background-color: #C8C6FA;

}

If you make the browser window narrow, you'll see that this gives us our entire mobile layout. Pretty easy, huh? No media queries required. That’s why it’s called “mobile-first”—the mobile version doesn’t require any special handling. Also notice that flex-wrap property in the containing .page div. This will make it very easy to implement our tablet and desktop layouts.



By keeping these base styles outside of the media queries, we’re able to override and add on to them as we implement our specific layouts. This is really convenient when, for instance, your designer wants to tweak the color scheme for the entire website. Instead of tracking down redundantbackground-color declarations in several @media rules, you only have to update it here. That change automatically applies to the mobile, tablet, and desktop layouts.

## tablet layout

On to the tablet layout. The only difference between the mobile and tablet mockups is that the **Sign Up** and **Feature** sections form a 2×2 grid instead of a single column.

Flexbox makes this real easy. Simply adjust the widths of the flex items to be half the screen and flex-wrap will take care of the rest. Of course, we only want this behavior to apply to tablet-sized screens, so it needs to go into an @media rule. Replace the existing /\* Tablet Styles \*/ media query with the following:

/\* Tablet Styles \*/

@**media** only screen and (min-width: 401px) and (max-width: 960px) {

.sign-up,

.feature-1,

.feature-2,

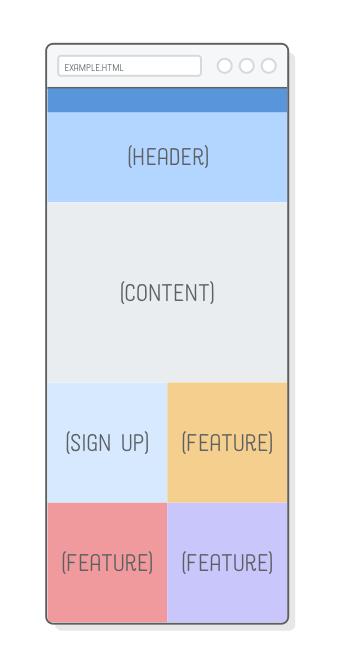
.feature-3 {

width: 50%;

}

}

To see these changes, make sure your browser window is between 400 pixels and 960 pixels wide, then scroll down to the bottom of the page. You should see a colorful grid:



Again, it doesn’t matter what the exact width of the screen is: this layout will fluidly respond to any width in the media query’s range. Our mobile layout is also fluid, so we now have a website that looks beautiful (if a bit empty) in every device smaller than 960px wide.

## desktop layout

And that’s where our desktop layout comes in. We don’t want our web page to expand endlessly, so we’re going to give it a fixed width and center it with auto-margins. As with tablet styles, this needs to go into a media query. Replace the existing /\* Desktop Styles \*/ media query with the following:

/\* Desktop Styles \*/

@**media** only screen and (min-width: 961px) {

.page {

width: 960px;

margin: 0 auto;

}

.feature-1,

.feature-2,

.feature-3 {

width: 33.3%;

}

.header {

height: 400px;

}

}

This gives us the correct widths for everything, and we have more real estate to play with, so we made the header a little taller, too. Almost there, but our desktop layout calls for some reordering: the **Sign Up** and **Content**boxes should appear underneath all the **Feature** sections.



This is where flexbox really shines. Trying to create this combination of mobile and desktop layouts would be very difficult with floats. With flexbox’s order property, it’s just a few lines of CSS. Append these rules to the desktop media query:

.sign-up {

height: 200px;

order: 1;

}

.content {

order: 2;

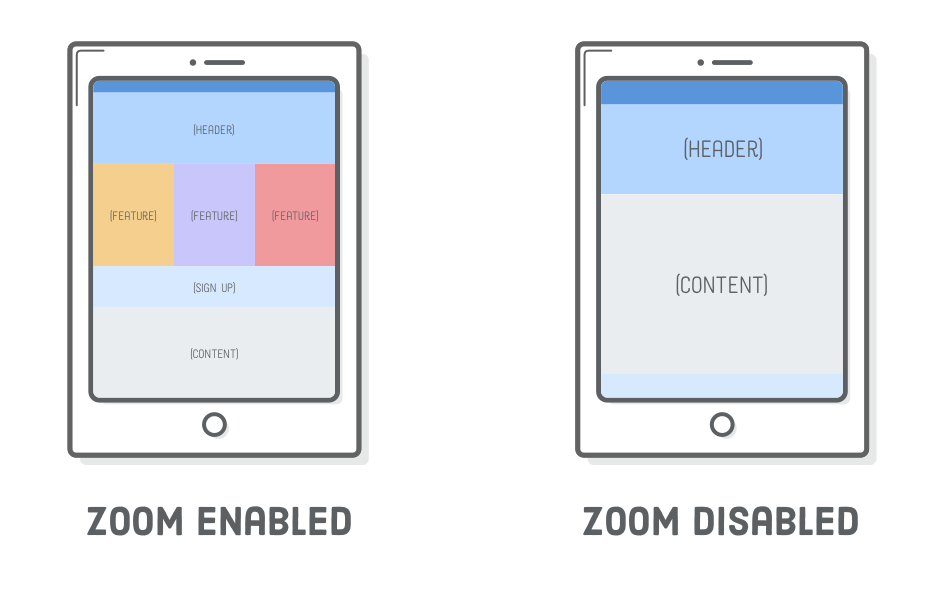
}

Ta da! A responsive website! Not bad for less than a hundred lines of CSS. More importantly, we didn’t have to alter a single line of HTML to accommodate our mobile, tablet, and desktop layouts.

This was just one example of laying out a responsive site. You can use these exact same techniques to implement all sorts of other designs. Start with the base styles that apply to your entire site, then tweak them for various device widths by selectively applying CSS rules with @media. You could even add another media query to, say, create a dedicated layout for ultra-widescreen monitors.

## disabling viewport zooming

We’ve got one final task for making a responsive web page. Before responsive design was a thing, mobile devices only had a desktop layout to work with. To cope with this, they zoomed out to fit the entire desktop layout into the width of the screen, letting the user interact with it by zooming in when necessary.

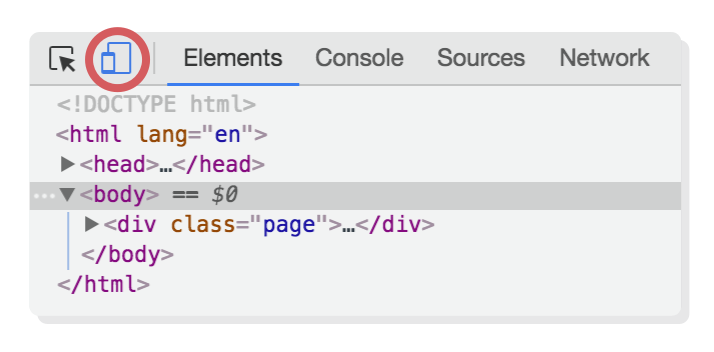


This default behavior will prevent mobile devices from using our mobile layout, which is obviously very terrible. To disable it, add the following element to the <head> of our document. Just like <meta charset='UTF-8'/>, this is a critical element that should be on every single web page you create:

<meta name='viewport'

content='width=device-width, initial-scale=1.0, maximum-scale=1.0' />

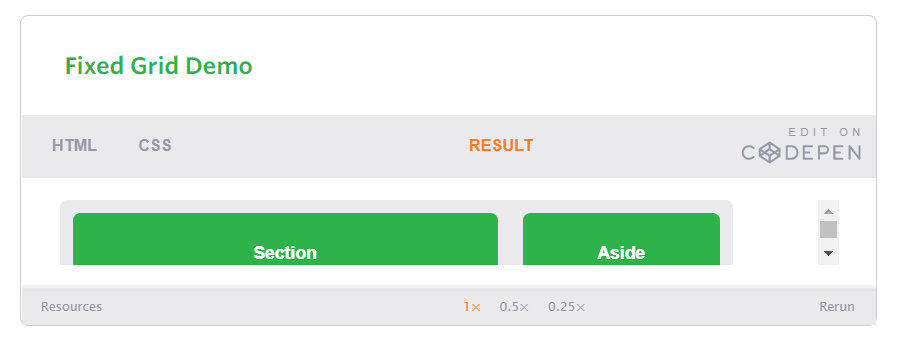
To see this in action, we’ll need to simulate a mobile device in our desktop browser. This is a little advanced for where we’re at right now, but we can give it a shot. Open up responsive.html in Google Chrome, then hit **View > Developer > Developer Tools** in the menu bar. Then, to simulate a mobile device, click the **Toggle Device Toolbar** icon, highlighted below.



You should see the zoom-disabled version of the above diagram in your browser, since it’s now pretending to be a mobile device. (We’ll save the in-depth discussion of Chrome dev tools for a future tutorial.)

Alternatively, if you’re reading this chapter on a smartphone, you can navigate to the live before and after versions of our example project to experience the effect of our viewport changes.

Example 1:



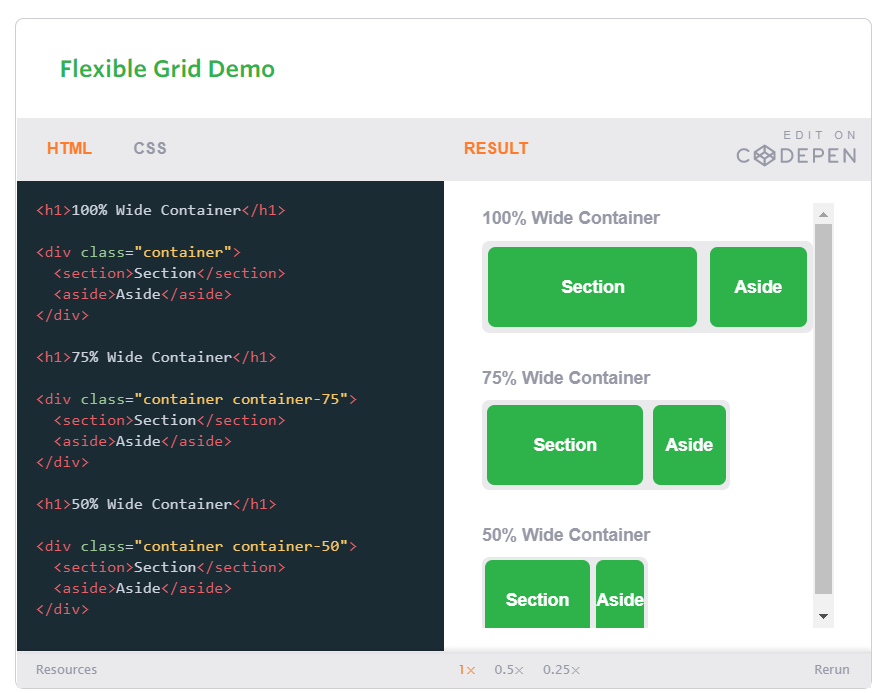
HTML

<div class="container">  
 <section>Section</section>  
 <aside>Aside</aside>  
</div>

CSS

body {  
 color: #fff;  
 font: 600 14px/24px "Open Sans", "HelveticaNeue-Light", "Helvetica Neue Light", "Helvetica Neue", Helvetica, Arial, "Lucida Grande", Sans-Serif;  
}  
.container:before,  
.container:after {  
 content: "";  
 display: table;  
}  
.container:after {  
 clear: both;  
}  
.container {  
 background: #eaeaed;  
 width: 538px;  
 \*zoom: 1;  
}  
.container,  
section,  
aside {  
 border-radius: 6px;  
}  
section,  
aside {  
 background: #2db34a;  
 margin: 10px;  
 padding: 20px 0;  
 text-align: center;  
}  
section {  
 float: left;  
 width: 340px;  
}  
aside {  
 float: right;  
 width: 158px;  
}

Example 2:



HTML

<h1>100% Wide Container</h1>  
  
<div class="container">  
 <section>Section</section>  
 <aside>Aside</aside>  
</div>  
  
<h1>75% Wide Container</h1>  
  
<div class="container container-75">  
 <section>Section</section>  
 <aside>Aside</aside>  
</div>  
  
<h1>50% Wide Container</h1>  
  
<div class="container container-50">  
 <section>Section</section>  
 <aside>Aside</aside>  
</div>

CSS

body {  
 font: 600 14px/24px "Open Sans", "HelveticaNeue-Light", "Helvetica Neue Light", "Helvetica Neue", Helvetica, Arial, "Lucida Grande", Sans-Serif;  
}  
h1 {  
 color: #9799a7;  
 font-size: 14px;  
 font-weight: bold;  
 margin-bottom: 6px;  
}  
.container:before,  
.container:after {  
 content: "";  
 display: table;  
}  
.container:after {  
 clear: both;  
}  
.container {  
 background: #eaeaed;  
 margin-bottom: 24px;  
 \*zoom: 1;  
}  
.container-75 {  
 width: 75%;  
}  
.container-50 {  
 margin-bottom: 0;  
 width: 50%;  
}  
.container,  
section,  
aside {  
 border-radius: 6px;  
}  
section,  
aside {  
 background: #2db34a;  
 color: #fff;  
 margin: 1.858736059%;  
 padding: 20px 0;  
 text-align: center;  
}  
section {  
 float: left;  
 width: 63.197026%;  
}  
aside {  
 float: right;  
 width: 29.3680297%;  
}

## Media Queries

Media queries were built as an extension to media types commonly found when targeting and including styles. Media queries provide the ability to specify different styles for individual browser and device circumstances, the width of the viewport or device orientation for example. Being able to apply uniquely targeted styles opens up a world of opportunity and leverage to responsive web design.

### Initializing Media Queries

There are a couple different ways to use media queries, using the @media rule inside of an existing style sheet, importing a new style sheet using the @import rule, or by linking to a separate style sheet from within the HTML document. Generally speaking it is recommend to use the @media rule inside of an existing style sheet to avoid any additional HTTP requests.

###### HTML

|  |  |
| --- | --- |
| 1  2  3 | <!-- Separate CSS File -->  <link href="styles.css" rel="stylesheet" media="all and (max-width: 1024px)"> |

###### CSS

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\* @media Rule \*/  @media all and (max-width: 1024px) {...}  /\* @import Rule \*/  @import url(styles.css) all and (max-width: 1024px) {...} |

Each media query may include a media type followed by one or more expressions. Common media types include all, screen, print, tv, and braille. The HTML5 specification includes new media types, even including 3d-glasses. Should a media type not be specified the media query will default the media type to screen.

The media query expression that follows the media type may include different media features and values, which then allocate to be true or false. When a media feature and value allocate to true, the styles are applied. If the media feature and value allocate to false the styles are ignored.

### Logical Operators in Media Queries

Logical operators in media queries help build powerful expressions. There are three different logical operators available for use within media queries, including and, not, and only.

Using the and logical operator within a media query allows an extra condition to be added, making sure that a browser or devices does both a, b, c, and so forth. Multiple individual media queries can be comma separated, acting as an unspoken or operator. The example below selects all media types between 800 and 1024 pixels wide.

|  |  |
| --- | --- |
| 1  2 | @media all and (min-width: 800px) and (max-width: 1024px) {...} |

The not logical operator negates the query, specifying any query but the one identified. In the example below the expression applies to any device that does not have a color screen. Black and white or monochrome screens would apply here for example.

|  |  |
| --- | --- |
| 1  2 | @media not screen and (color) {...} |

The only logical operator is a new operator and is not recognized by user agents using the HTML4 algorithm, thus hiding the styles from devices or browsers that don’t support media queries. Below, the expression selects only screens in a portrait orientation that have a user agent capable of rending media queries.

|  |  |
| --- | --- |
| 1  2 | @media only screen and (orientation: portrait) {...} |

#### Omitting a Media Type

When using the not and only logical operators the media type may be left off. In this case the media type is defaulted to all.

### Media Features in Media Queries

Knowing the media query syntax and how logical operators work is a great introduction to media queries but the true work comes with media features. Media features identify what attributes or properties will be targeted within the media query expression.

#### Height & Width Media Features

One of the most common media features revolves around determining a height or width for a device or browser viewport. The height and width may be found by using the height and width media features. Each of these media features may then also be prefixed with the min or max qualifiers, building a feature such as min-width or max-width.

The height and width features are based off the height and width of the viewport rendering area, the browser window for example. Values for these height and width media features may be any length unit, relative or absolute.

|  |  |
| --- | --- |
| 1  2 | @media all and (min-width: 320px) and (max-width: 780px) {...} |

Within responsive design the most commonly used features include min-width and max-width. These help build responsive websites on desktops and mobile devices equally, avoiding any confusion with device features.

#### Using Minimum & Maximum Prefixes

The min and max prefixes can be used on quite a few media features. The minprefix indicates a values of greater than or equal to while the max prefix indicates a value of less than or equal to. Using min and max prefixes avoid any conflict with the general HTML syntax, specifically not using the < and >symbols.

#### Orientation Media Feature

The orientation media feature determines if a device is in the landscape or portraitorientation. The landscape mode is triggered when the display is wider than taller, and the portrait mode is triggered when the display is taller than wider. This media feature plays a large part with mobile devices.

|  |  |
| --- | --- |
| 1  2 | @media all and (orientation: landscape) {...} |

#### Aspect Ratio Media Features

The aspect-ratio and device-aspect-ratio features specifies the width/heightpixel ratio of the targeted rendering area or output device. The min and max prefixes are available to use with the different aspect ratio features, identifying a ratio above or below that of which is stated.

The value for the aspect ratio feature consist of two positive integers separated by a forward slash. The first integer identifies the width in pixels while the second integer identifies the height in pixels.

|  |  |
| --- | --- |
| 1  2 | @media all and (min-device-aspect-ratio: 16/9) {...} |

#### Pixel Ratio Media Features

In addition to the aspect ratio media features there are also pixel-ratiomedia features. These features do include the device-pixel-ratio feature as well as min and max prefixes. Specifically, the pixel ratio feature is great for identifying high definition devices, including retina displays. Media queries for doing so look like the following.

|  |  |
| --- | --- |
| 1  2 | @media only screen and (-webkit-min-device-pixel-ratio: 1.3), only screen and (min-device-pixel-ratio: 1.3) {...} |

#### Resolution Media Feature

The resolution media feature specifies the resolution of the output device in pixel density, also known as dots per inch or DPI. The resolution media feature does accept the min and max prefixes. Additionally, the resolution media feature will accept dots per pixel (1.3dppx), dots per centimeter (118dpcm), and other length based resolution values.

|  |  |
| --- | --- |
| 1  2 | @media print and (min-resolution: 300dpi) {...} |

#### Other Media Features

Other media features include identifying available output colors with use of the color, color-index, and monochrome features, identifying bitmap devices with the gridfeature, and identifying the scanning process of a television with the scan feature. These features are less common but equally as helpful when needed.

#### Media Query Browser Support

Unfortunately media queries do not work within Internet Explorer 8 and below, as well as other legacy browsers. There are, however, a couple suitable polyfills written in Javascript.

Respond.js is a lightweight polyfill that only looks for min/max-width media types, which is perfect should those be the only media query types used. CSS3-MediaQueries.js is a more developed, and heavier, polyfill offering support for a larger array of more complex media queries. Additionally, keep in mind any polyfill can have performance concerns, and potentially slow down websites. Make sure that any given polyfill is worth the performance trade off.

### Media Queries Demo

Using media queries we will now rewrite the flexible layout we built previously. One of the current problems within the demo appears when the aside width becomes uselessly small within smaller viewports. Adding a media query for viewports under 420 pixels wide we can change the layout by turning off the floats and changing the widths of the section and aside.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @media all and (max-width: 420px) {  section, aside {  float: none;  width: auto;  }  } |

## Mobile First

One popular technique with using media queries is called mobile first. The mobile firstapproach includes using styles targeted at smaller viewports as the default styles for a website, then use media queries to add styles as the viewport grows.

The operating belief behind mobile first design is that a user on a mobile device, commonly using a smaller viewport, shouldn’t have to load the styles for a desktop computer only to have them over written with mobile styles later. Doing so is a waste of bandwidth. Bandwidth that is precious to any users looking for a snappy website.

The mobile first approach also advocates designing with the constraints of a mobile user in mind. Before too long, the majority of Internet consumption will be done on a mobile device. Plan for them accordingly and develop intrinsic mobile experiences.

A breakout of mobile first media queries might look at bit like the following.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\* Default styles first then media queries \*/  @media screen and (min-width: 400px) {...}  @media screen and (min-width: 600px) {...}  @media screen and (min-width: 1000px) {...}  @media screen and (min-width: 1400px) {...} |

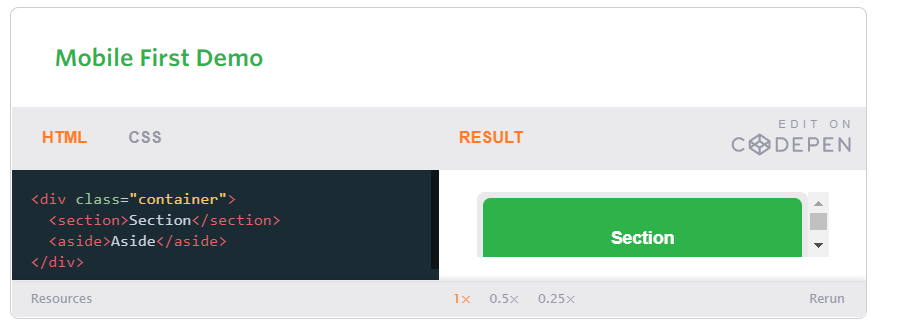
Additionally, downloading unnecessary media assets can be stopped by using media queries. Generally speaking, avoiding CSS3 shadows, gradients, transforms, and animations within mobile styles isn’t a bad idea either. When used excessively, they cause heavy loading and can even reduce a device’s battery life.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | /\* Default media \*/  body {  background: #ddd;  }  /\* Media for larger devices \*/  @media screen and (min-width: 800px) {  body {  background-image: url("bg.png") 50% 50% no-repeat;  }  } |

### Mobile First Demo

Adding media queries to our previous example, we overwrote a handful of styles in order to have a better layout on viewports under 420 pixels wide. Rewriting this code to use the mobile styles first by default then adding media queries to adjust for viewports over 420 pixels wide we build the following:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | section,  aside {  margin: 1.858736059%;  }  @media all and (min-width: 420px) {  .container {  max-width: 538px;  }  section {  float: left;  width: 63.197026%;  }  aside {  float: right;  width: 29.3680297%;  }  } |



HTML

<div class="container">  
 <section>Section</section>  
 <aside>Aside</aside>  
</div>

body {  
 font: 600 14px/24px "Open Sans", "HelveticaNeue-Light", "Helvetica Neue Light", "Helvetica Neue", Helvetica, Arial, "Lucida Grande", Sans-Serif;  
}  
.container:before,  
.container:after {  
 content: "";  
 display: table;  
}  
.container:after {  
 clear: both;  
}  
.container {  
 background: #eaeaed;  
 \*zoom: 1;  
}  
.container,  
section,  
aside {  
 border-radius: 6px;  
}  
section,  
aside {  
 background: #2db34a;  
 color: #fff;  
 margin: 1.858736059%;  
 padding: 20px 0;  
 text-align: center;  
}  
@media all and (min-width: 420px) {  
 .container {  
 max-width: 538px;  
 }  
 section {  
 float: left;  
 width: 63.197026%;  
 }  
 aside {  
 float: right;  
 width: 29.3680297%;  
 }  
}

#### Viewport Height & Width

Using the viewport meta tag with either the height or width values will define the height or width of the viewport respectively. Each value accepts either a positive integer or keyword. For the height property the keyword device-height value is accepted, and for the width property the keyword device-width is accepted. Using these keywords will inherit the device’s default height and width value.

For the best results, and the best looking website, it is recommend that you use the device defaults by applying the device-height and device-width values.

<meta name="viewport" content="width=device-width">



#### Viewport Scale

To control how a website is scaled on a mobile device, and how users can continue to scale a website, use the minimum-scale, maximum-scale, initial-scale, and user-scalable properties.

The initial-scale of a website should be set to 1 as this defines the ratio between the device height, while in a portrait orientation, and the viewport size. Should a device be in landscape mode this would be the ratio between the device width and the viewport size. Values for initial-scale should always be a positive integer between 0and 10.

|  |  |
| --- | --- |
| 1  2 | <meta name="viewport" content="initial-scale=2"> |



The minimum-scale and maximum-scale values determine how small and how large a viewport may be scaled. When using minimum-scale the value should be a positive integer lower than or equal to the initial-scale. Using the same reasoning, the maximum-scale value should be a positive integer greater than or equal to the initial-scale. Values for both of these must also be between 0 and 10.

|  |  |
| --- | --- |
| 1  2 | <meta name="viewport" content="minimum-scale=0"> |

Generally speaking, these values should not be set to the same value as the initial-scale. This would disable any zooming, which can be accomplished instead by using the user-scalable value. Setting the user-scalable value to no will disable any zooming. Alternatively, setting the user-scalable value to yes will turn on zooming.

Turning off the ability to scale a website is a **bad idea**. It harms accessibility and usability, preventing those with disabilities from viewing a website as desired.

|  |  |
| --- | --- |
| 1  2 | <meta name="viewport" content="user-scalable=yes"> |

#### Viewport Resolution

Letting the browser decide how to scale a website based off any viewport scale values usually does the trick. When more control is needed, specifically over the resolution of a device, the target-densitydpi value may be used. The target-densitydpi viewport accepts a handful of values including device-dpi, high-dpi, medium-dpi, low-dpi, or an actual DPI number.

Using the target-densitydpi viewport value is rare, but extremely helpful when pixel by pixel control is needed.

|  |  |
| --- | --- |
| 1  2 | <meta name="viewport" content="target-densitydpi=device-dpi"> |

#### Combining Viewport Values

The viewport meta tag will accept individual values as well as multiple values, allowing multiple viewport properties to be set at once. Setting multiple values requires comma separating them within the content attribute value. One of the recommended viewport values is outlined below, using both the width and initial-scale properties.

|  |  |
| --- | --- |
| 1  2 | <meta name="viewport" content="width=device-width, initial-scale=1"> |



#### CSS Viewport Rule

Since the viewport meta tag revolves so heavily around setting the styles of how a website should be rendered it has been recommend to move the viewport from a meta tag with HTML to an @ rule within CSS. This helps keep the style separated from content, providing a more semantic approach.

Currently some browsers have already implemented the @viewport rule, however support isn’t great across the board. The previously recommended viewport meta tag would look like the following @viewport rule in CSS.

|  |  |
| --- | --- |
| 1  2  3  4  5 | @viewport {  width: device-width;  zoom: 1;  } |

## Flexible Media

The final, equally important aspect to responsive web design involves flexible media. As viewports begin to change size media doesn’t always follow suit. Images, videos, and other media types need to be scalable, changing their size as the size of the viewport changes.

One quick way to make media scalable is by using the max-width property with a value of 100%. Doing so ensures that as the viewport gets smaller any media will scale down according to its containers width.

|  |  |
| --- | --- |
| 1  2  3  4 | img, video, canvas {  max-width: 100%;  } |

<h1>100% Wide Container</h1>  
  
<figure>  
 <img src="https://s3-us-west-2.amazonaws.com/s.cdpn.io/29841/chicago.jpg" alt="Chicago">  
</figure>  
  
<h1>75% Wide Container</h1>  
  
 <figure class="figure-75">  
 <img src="https://s3-us-west-2.amazonaws.com/s.cdpn.io/29841/chicago.jpg" alt="Chicago">  
</figure>  
  
 <h1>50% Wide Container</h1>  
  
 <figure class="figure-50">  
 <img src="https://s3-us-west-2.amazonaws.com/s.cdpn.io/29841/chicago.jpg" alt="Chicago">  
</figure>

body {  
 font: 600 14px/24px "Open Sans", "HelveticaNeue-Light", "Helvetica Neue Light", "Helvetica Neue", Helvetica, Arial, "Lucida Grande", Sans-Serif;  
}  
h1 {  
 color: #9799a7;  
 font-size: 14px;  
 font-weight: bold;  
 margin-bottom: 6px;  
}  
figure {  
 margin: 0 0 24px 0;  
}  
.figure-75 {  
 width: 75%;  
}  
.figure-50 {  
 margin-bottom: 0;  
 width: 50%;  
}  
img {  
 display: block;  
 max-width: 100%;  
}