Front End Libraries Notes

Patrick Adams

November 1, 2020

Contents

1 Bootstrap

1.1 Use Responsive Design with Bootstrap Fluid Containers

In the HTML5 and CSS section of freeCodeCamp we built a Cat Photo App. Now let's go back to it. This time, we'll style it using the popular Bootstrap responsive CSS framework.
Bootstrap will figure out how wide your screen is and respond by resizing your HTML elements - hence the name responsive design.
With responsive design, there is no need to design a mobile version of your website. It will look good on devices with screens of any width.

You can add Bootstrap to any app by adding the following code to the top of your HTML:
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous"/>
In this case, we've already added it for you to this page behind the scenes. Note that using either > or /> to close the link tag is acceptable.
To get started, we should nest all of our HTML (except the link tag and the style element) in a div element with the class container-fluid.

1.2 Make Images Mobile Responsive

First, add a new image below the existing one. Set its src attribute to https://bit.ly/fcc-running-cats.
It would be great if this image could be exactly the width of our phone's screen.
Fortunately, with Bootstrap, all we need to do is add the img-responsive class to your image. Do this, and the image should perfectly fit the width of your page.

1.3 Center Text with Bootstrap

Now that we're using Bootstrap, we can center our heading element to make it look better. All we need to do is add the class text-center to our h2 element.
Remember that you can add several classes to the same element by separating each of them with a space, like this:
<h2 class="red-text text-center">your text</h2>

1.4 Create a Bootstrap Button

Bootstrap has its own styles for button elements, which look much better than the plain HTML ones.
Create a new button element below your large kitten photo. Give it the btn and btn-default classes, as well as the text of "Like".

1.5 Create a Block Element Bootstrap Button

Normally, your button elements with the btn and btn-default classes are only as wide as the text that they contain. For example:
<button class="btn btn-default">Submit</button>
This button would only be as wide as the word "Submit".
Submit
By making them block elements with the additional class of btn-block, your button will stretch to fill your page's entire horizontal space and any elements following it will flow onto a "new line" below the block.
<button class="btn btn-default btn-block">Submit</button>
This button would take up 100% of the available width.
Submit
Note that these buttons still need the btn class.
Add Bootstrap's btn-block class to your Bootstrap button.

1.6 Taste the Bootstrap Button Color Rainbow

The btn-primary class is the main color you'll use in your app. It is useful for highlighting actions you want your user to take.
Replace Bootstrap's btn-default class with btn-primary in your button.

Note that this button will still need the btn and btn-block classes.

1.7 Call out Optional Actions with btn-info

Bootstrap comes with several pre-defined colors for buttons. The btn-info class is used to call attention to optional actions that the user can take.
Create a new block-level Bootstrap button below your "Like" button with the text "Info", and add Bootstrap's btn-info and btn-block classes to it.
Note that these buttons still need the btn and btn-block classes.

1.8 Warn Your Users of a Dangerous Action with btn-danger

Bootstrap comes with several pre-defined colors for buttons. The btn-danger class is the button color you'll use to notify users that the button performs a destructive action, such as deleting a cat photo.
Create a button with the text "Delete" and give it the class btn-danger.
Note that these buttons still need the btn and btn-block classes.

1.9 Use the Bootstrap Grid to Put Elements Side By Side

Bootstrap uses a responsive 12-column grid system, which makes it easy to put elements into rows and specify each element's relative width. Most of Bootstrap's classes can be applied to a div element.
Bootstrap has different column width attributes that it uses depending on how wide the user's screen is. For example, phones have narrow screens, and laptops have wider screens.
Take for example Bootstrap's col-md-* class. Here, md means medium, and * is a number specifying how many columns wide the element should be. In this case, the column width of an element on a medium-sized screen, such as a laptop, is being specified.
In the Cat Photo App that we're building, we'll use col-xs-*, where xs means extra small (like an extra-small mobile phone screen), and * is the number of columns specifying how many columns wide the element should be.
Put the Like, Info and Delete buttons side-by-side by nesting all three of them within one <div class="row"> element, then each of them within a <div class="col-xs-4"> element.
The row class is applied to a div, and the buttons themselves can be nested within it.

1.10 Ditch Custom CSS for Bootstrap

We can clean up our code and make our Cat Photo App look more conventional by using Bootstrap's built-in styles instead of the custom styles we created earlier.
Don't worry - there will be plenty of time to customize our CSS later.
Delete the .red-text, p, and .smaller-image CSS declarations from your style element so that the only declarations left in your style element are h2 and thick-green-border.
Then delete the p element that contains a dead link. Then remove the red-text class from your h2 element and replace it with the text-primary Bootstrap class.
Finally, remove the "smaller-image" class from your first img element and replace it with the img-responsive class.

1.11 Use a span to Target Inline Elements

You can use spans to create inline elements. Remember when we used the btn-block class to make the button fill the entire row?
normal button

btn-block button
That illustrates the difference between an "inline" element and a "block" element.
By using the inline span element, you can put several elements on the same line, and even style different parts of the same line differently.
Nest the word "love" in your "Things cats love" element below within a span element. Then give that span the class text-danger to make the text red.
Here's how you would do this with the "Top 3 things cats hate" element:
<p>Top 3 things cats <span class="text-danger">hate:</span></p>

1.12 Create a Custom Heading

We will make a simple heading for our Cat Photo App by putting the title and relaxing cat image in the same row.
Remember, Bootstrap uses a responsive grid system, which makes it easy to put elements into rows and specify each element's relative width. Most of Bootstrap's classes can be applied to a div element.
Nest your first image and your h2 element within a single <div class="row"> element. Nest your h2 element within a <div class="col-xs-8"> and your image in a <div class="col-xs-4"> so that they are on the same line.
Notice how the image is now just the right size to fit along the text?

1.13 Add Font Awesome Icons to our Buttons

Font Awesome is a convenient library of icons. These icons can be webfonts or vector graphics. These icons are treated just like fonts. You can specify their size using pixels, and they will assume the font size of their parent HTML elements.
You can include Font Awesome in any app by adding the following code to the top of your HTML:
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.8.1/css/all.css" integrity="sha384-50oBUHEmvpQ+1l
crossorigin="anonymous">
In this case, we've already added it for you to this page behind the scenes.
The i element was originally used to make other elements italic, but is now commonly used for icons. You can add the Font Awesome classes to the i element to turn it into an icon, for example:
<i class="fas fa-info-circle"></i>
Note that the span element is also acceptable for use with icons.

1.14 Add Font Awesome Icons to all of our Buttons

Font Awesome is a convenient library of icons. These icons can be web fonts or vector graphics. These icons are treated just like fonts. You can specify their size using pixels, and they will assume the font size of their parent HTML elements.

1.15 Responsively Style Radio Buttons

You can use Bootstrap's col-xs-* classes on form elements, too! This way, our radio buttons will be evenly spread out across the page, regardless of how wide the screen resolution is.
Nest both your radio buttons within a <div class="row"> element. Then nest each of them within a <div class="col-xs-6"> element.
Note: As a reminder, radio buttons are input elements of type radio.

1.16 Responsively Style Checkboxes

Since Bootstrap's col-xs-* classes are applicable to all form elements, you can use them on your checkboxes too! This way, the checkboxes will be evenly spread out across the page, regardless of how wide the screen resolution is.

1.17 Style Text Inputs as Form Controls

You can add the fa-paper-plane Font Awesome icon by adding <i class="fa fa-paper-plane"></i> within your submit button element.
Give your form's text input field a class of form-control. Give your form's submit button the classes btn btn-primary. Also give this button the Font Awesome icon of fa-paper-plane.
All textual <input>, <textarea>, and <select> elements with the class .form-control have a width of 100%.

1.18 Line up Form Elements Responsively with Bootstrap

Now let's get your form input and your submission button on the same line. We'll do this the same way we have previously: by using a div element with the class row, and other div elements within it using the col-xs-* class.
Nest both your form's text input and submit button within a div with the class row. Nest your form's text input within a div with the class of col-xs-7. Nest your form's submit button in a div with the class col-xs-5.
This is the last challenge we'll do for our Cat Photo App for now. We hope you've enjoyed learning Font Awesome, Bootstrap, and responsive design!

1.19 Create a Bootstrap Headline

Now let's build something from scratch to practice our HTML, CSS and Bootstrap skills.
We'll build a jQuery playground, which we'll soon put to use in our jQuery challenges.
To start with, create an h3 element, with the text jQuery Playground.
Color your h3 element with the text-primary Bootstrap class, and center it with the text-center Bootstrap class.

1.20 House our page within a Bootstrap container-fluid div

Now let's make sure all the content on your page is mobile-responsive.
Let's nest your h3 element within a div element with the class container-fluid.

1.21 Create a Bootstrap Row

Now we'll create a Bootstrap row for our inline elements.
Create a div element below the h3 tag, with a class of row.

1.22 Split Your Bootstrap Row

Now that we have a Bootstrap Row, let's split it into two columns to house our elements.
Create two div elements within your row, both with the class col-xs-6.

1.23 Create Bootstrap Wells

Bootstrap has a class called well that can create a visual sense of depth for your columns.
Nest one div element with the class well within each of your col-xs-6 div elements.

1.24 Add Elements within Your Bootstrap Wells

Now we're several div elements deep on each column of our row. This is as deep as we'll need to go. Now we can add our button elements.
Nest three button elements within each of your well div elements.

1.25 Apply the Default Bootstrap Button Style

Bootstrap has another button class called btn-default.
Apply both the btn and btn-default classes to each of your button elements.

1.26 Create a Class to Target with jQuery Selectors

Not every class needs to have corresponding CSS. Sometimes we create classes just for the purpose of selecting these elements more easily using jQuery.
Give each of your button elements the class target.

1.27 Add id Attributes to Bootstrap Elements

Recall that in addition to class attributes, you can give each of your elements an id attribute.
Each id must be unique to a specific element and used only once per page.
Let's give a unique id to each of our div elements of class well.
Remember that you can give an element an id like this:
<div class="well" id="center-well">
Give the well on the left the id of left-well. Give the well on the right the id of right-well.

1.28 Label Bootstrap Wells

For the sake of clarity, let's label both of our wells with their ids.
Above your left-well, inside its col-xs-6 div element, add a h4 element with the text #left-well.
Above your right-well, inside its col-xs-6 div element, add a h4 element with the text #right-well.

1.29 Give Each Element a Unique id

We will also want to be able to use jQuery to target each button by its unique id.
Give each of your buttons a unique id, starting with target1 and ending with target6.
Make sure that target1 to target3 are in #left-well, and target4 to target6 are in #right-well.

1.30 Label Bootstrap Buttons

Just like we labeled our wells, we want to label our buttons.
Give each of your button elements text that corresponds to its id's selector.

1.31 Use Comments to Clarify Code

When we start using jQuery, we will modify HTML elements without needing to actually change them in HTML.
Let's make sure that everyone knows they shouldn't actually modify any of this code directly.
Remember that you can start a comment with <!-- and end a comment with -->
Add a comment at the top of your HTML that says Code below this line should not be changed

2 Jquery

2.1 Learn How Script Tags and Document Ready Work

Now we're ready to learn jQuery, the most popular JavaScript tool of all time.
Before we can start using jQuery, we need to add some things to our HTML.
First, add a script element at the top of your page. Be sure to close it on the following line.
Your browser will run any JavaScript inside a script element, including jQuery.
Inside your script element, add this code: $(document).ready(function() { to your script. Then close it on the following line (still inside your script element) with: });
We'll learn more about functions later. The important thing to know is that code you put inside this function will run as soon as your browser has loaded your page.
This is important because without your document ready function, your code may run before your HTML is rendered, which would cause bugs.

2.2 Target HTML Elements with Selectors Using jQuery

Now we have a document ready function.
Now let's write our first jQuery statement. All jQuery functions start with a $, usually referred to as a dollar sign operator, or as bling.
jQuery often selects an HTML element with a selector, then does something to that element.
For example, let's make all of your button elements bounce. Just add this code inside your document ready function:
$("button").addClass("animated bounce");
Note that we've already included both the jQuery library and the Animate.css library in the background so that you can use them in the editor. So you are using jQuery to apply the Animate.css bounce class to your button elements.

2.3 Target Elements by Class Using jQuery

You see how we made all of your button elements bounce? We selected them with $("button"), then we added some CSS classes to them with .addClass("animated bounce");.
You just used jQuery's .addClass() function, which allows you to add classes to elements.
First, let's target your div elements with the class well by using the $(".well") selector.
Note that, just like with CSS declarations, you type a . before the class's name.
Then use jQuery's .addClass() function to add the classes animated and shake.
For example, you could make all the elements with the class text-primary shake by adding the following to your document ready function:
$(".text-primary").addClass("animated shake");

2.4 Target Elements by id Using jQuery

You can also target elements by their id attributes.
First target your button element with the id target3 by using the $("#target3") selector.
Note that, just like with CSS declarations, you type a # before the id's name.
Then use jQuery's .addClass() function to add the classes animated and fadeOut.
Here's how you'd make the button element with the id target6 fade out:
$("#target6").addClass("animated fadeOut").

2.5 Delete Your jQuery Functions

These animations were cool at first, but now they're getting kind of distracting.
Delete all three of these jQuery functions from your document ready function, but leave your document ready function itself intact.

2.6 Target the Same Element with Multiple jQuery Selectors

Now you know three ways of targeting elements: by type: $("button"), by class: $(".btn"), and by id $("#target1").
Although it is possible to add multiple classes in a single .addClass() call, let's add them to the same element in three separate ways.
Using .addClass(), add only one class at a time to the same element, three different ways:
Add the animated class to all elements with type button.
Add the shake class to all the buttons with class .btn.
Add the btn-primary class to the button with id #target1.
NoteYou should only be targeting one element and adding only one class at a time. Altogether, your three individual selectors will end up adding the three classes shake, animated, and btn-primary to #target1.

2.7 Remove Classes from an Element with jQuery

In the same way you can add classes to an element with jQuery's addClass() function, you can remove them with jQuery's removeClass() function.
Here's how you would do this for a specific button:
$("#target2").removeClass("btn-default");
Let's remove the btn-default class from all of our button elements.

2.8 Change the CSS of an Element Using jQuery

We can also change the CSS of an HTML element directly with jQuery.
jQuery has a function called .css() that allows you to change the CSS of an element.
Here's how we would change its color to blue:
$("#target1").css("color", "blue");
This is slightly different from a normal CSS declaration, because the CSS property and its value are in quotes, and separated with a comma instead of a colon.
Delete your jQuery selectors, leaving an empty document ready function.
Select target1 and change its color to red.

2.9 Disable an Element Using jQuery

You can also change the non-CSS properties of HTML elements with jQuery. For example, you can disable buttons.
When you disable a button, it will become grayed-out and can no longer be clicked.
jQuery has a function called .prop() that allows you to adjust the properties of elements.
Here's how you would disable all buttons:
$("button").prop("disabled", true);
Disable only the target1 button.

2.10 Change Text Inside an Element Using jQuery

Using jQuery, you can change the text between the start and end tags of an element. You can even change HTML markup.
jQuery has a function called .html() that lets you add HTML tags and text within an element. Any content previously within the element will be completely replaced with the content you provide using this function.
Here's how you would rewrite and emphasize the text of our heading:
$("h3").html("<em>jQuery Playground</em>");
jQuery also has a similar function called .text() that only alters text without adding tags. In other words, this function will not evaluate any HTML tags passed to it, but will instead treat it as the text you want to replace the existing content with.
Change the button with id target4 by emphasizing its text.
View our news article for <em> to learn the difference between <i> and <em> and their uses.
Note that while the <i> tag has traditionally been used to emphasize text, it has since been adopted for use as a tag for icons. The <em> tag is now widely accepted as the tag for emphasis. Either will work for this challenge.

2.11 Remove an Element Using jQuery

Now let's remove an HTML element from your page using jQuery.
jQuery has a function called .remove() that will remove an HTML element entirely
Remove element target4 from the page by using the .remove() function.

2.12 Use appendTo to Move Elements with jQuery

Now let's try moving elements from one div to another.
jQuery has a function called appendTo() that allows you to select HTML elements and append them to another element.
For example, if we wanted to move target4 from our right well to our left well, we would use:
$("#target4").appendTo("#left-well");
Move your target2 element from your left-well to your right-well.

2.13 Clone an Element Using jQuery

In addition to moving elements, you can also copy them from one place to another.
jQuery has a function called clone() that makes a copy of an element.
For example, if we wanted to copy target2 from our left-well to our right-well, we would use:
$("#target2").clone().appendTo("#right-well");
Did you notice this involves sticking two jQuery functions together? This is called function chaining and it's a convenient way to get things done with jQuery.
Clone your target5 element and append it to your left-well.

2.14 Target the Parent of an Element Using jQuery

Every HTML element has a parent element from which it inherits properties.
For example, your jQuery Playground h3 element has the parent element of <div class="container-fluid">, which itself has the parent body.
jQuery has a function called parent() that allows you to access the parent of whichever element you've selected.
Here's an example of how you would use the parent() function if you wanted to give the parent element of the left-well element a background color of blue:

$("#left-well").parent().css("background-color", "blue")
Give the parent of the #target1 element a background-color of red.

2.15 Target the Children of an Element Using jQuery

When HTML elements are placed one level below another they are called children of that element. For example, the button elements in this challenge with the text "#target1", "#target2", and "#target3" are all children of the <div class="well" id="left-well"> element.
jQuery has a function called children() that allows you to access the children of whichever element you've selected.
Here's an example of how you would use the children() function to give the children of your left-well element the color blue:
$("#left-well").children().css("color", "blue")

2.16 Target a Specific Child of an Element Using jQuery

You've seen why id attributes are so convenient for targeting with jQuery selectors. But you won't always have such neat ids to work with.
Fortunately, jQuery has some other tricks for targeting the right elements.
jQuery uses CSS Selectors to target elements. The target:nth-child(n) CSS selector allows you to select all the nth elements with the target class or element type.
Here's how you would give the third element in each well the bounce class:
$(".target:nth-child(3)").addClass("animated bounce");
Make the second child in each of your well elements bounce. You must select the elements' children with the target class.

2.17 Target Even Elements Using jQuery

You can also target elements based on their positions using :odd or :even selectors.
Note that jQuery is zero-indexed which means the first element in a selection has a position of 0. This can be a little confusing as, counter-intuitively, :odd selects the second element (position 1), fourth element (position 3), and so on.
Here's how you would target all the odd elements with class target and give them classes:
$(".target:odd").addClass("animated shake");
Try selecting all the even target elements and giving them the classes of animated and shake. Remember that even refers to the position of elements with a zero-based system in mind.

2.18 Use jQuery to Modify the Entire Page

We're done playing with our jQuery playground. Let's tear it down!
jQuery can target the body element as well.
Here's how we would make the entire body fade out: $("body").addClass("animated fadeOut");
But let's do something more dramatic. Add the classes animated and hinge to your body element.

3 Sass

3.1 Store Data with Sass Variables

One feature of Sass that’s different than CSS is it uses variables. They are declared and set to store data, similar to JavaScript.
In JavaScript, variables are defined using the let and const keywords. In Sass, variables start with a $ followed by the variable name.
Here are a couple examples:
“‘scss
```
$main-fonts: Arial, sans-serif;
$headings-color: green;
//To use variables:
h1 {
font-family: $main-fonts;
color: $headings-color;
}
```
“‘
One example where variables are useful is when a number of elements need to be the same color. If that color is changed, the only place to edit the code is the variable value.

3.2 Nest CSS with Sass

Sass allows nesting of CSS rules, which is a useful way of organizing a style sheet.
Normally, each element is targeted on a different line to style it, like so:
“‘scss
```
nav {
background-color: red;
}
nav ul {
list-style: none;
}
nav ul li {
display: inline-block;
}
```
“‘
For a large project, the CSS file will have many lines and rules. This is where nesting can help organize your code by placing child style rules within the respective parent elements:
“‘scss
```
nav {
background-color: red;
ul {
list-style: none;
li {
display: inline-block;
}
}
}
```
“‘

3.3 Create Reusable CSS with Mixins

In Sass, a mixin is a group of CSS declarations that can be reused throughout the style sheet.
Newer CSS features take time before they are fully adopted and ready to use in all browsers. As features are added to browsers, CSS rules using them may need vendor prefixes. Consider "box-shadow":

```scss
div {
-webkit-box-shadow: 0px 0px 4px #fff;
-moz-box-shadow: 0px 0px 4px #fff;
-ms-box-shadow: 0px 0px 4px #fff;
box-shadow: 0px 0px 4px #fff;
}
```

It's a lot of typing to re-write this rule for all the elements that have a box-shadow, or to change each value to test different effects.
Mixins are like functions for CSS. Here is how to write one:

```scss
@mixin box-shadow($x, $y, $blur, $c){
-webkit-box-shadow: $x $y $blur $c;
-moz-box-shadow: $x $y $blur $c;
-ms-box-shadow: $x $y $blur $c;
box-shadow: $x $y $blur $c;
}
```

The definition starts with @mixin followed by a custom name. The parameters (the $x, $y, $blur, and $c in the example above) are optional.
Now any time a box-shadow rule is needed, only a single line calling the mixin replaces having to type all the vendor prefixes. A mixin is called with the @include directive:

```scss
div {
@include box-shadow(0px, 0px, 4px, #fff);
}
```

3.4 Use @if and @else to Add Logic To Your Styles

The @if directive in Sass is useful to test for a specific case - it works just like the if statement in JavaScript.

```scss
@mixin make-bold($bool) {
@if $bool == true {
font-weight: bold;
}
}
```

And just like in JavaScript, @else if and @else test for more conditions:

```scss
@mixin text-effect($val) {
@if $val == danger {
color: red;
}
@else if $val == alert {
color: yellow;
}
@else if $val == success {
```

```
color: green;
}
@else {
color: black;
}
}
```
“‘

3.5 Use @for to Create a Sass Loop

The @for directive adds styles in a loop, very similar to a for loop in JavaScript.
@for is used in two ways: "start through end" or "start to end". The main difference is that the "start to end" excludes the end number as part of the count, and "start through end" includes the end number as part of the count.
Here's a start through end example:
“‘scss
```
@for $i from 1 through 12 {
.col-#{$i} { width: 100%/12 * $i; }
}
```
“‘
The #{$i} part is the syntax to combine a variable (i) with text to make a string. When the Sass file is converted to CSS, it looks like this:
“‘scss
```
.col-1 {
width: 8.33333%;
}
.col-2 {
width: 16.66667%;
}
...
.col-12 {
width: 100%;
}
```
“‘
This is a powerful way to create a grid layout. Now you have twelve options for column widths available as CSS classes.

3.6 Use @each to Map Over Items in a List

The last challenge showed how the @for directive uses a starting and ending value to loop a certain number of times. Sass also offers the @each directive which loops over each item in a list or map.
On each iteration, the variable gets assigned to the current value from the list or map.
“‘scss
```
@each $color in blue, red, green {
.#{$color}-text {color: $color;}
}
```
“‘
A map has slightly different syntax. Here's an example:
“‘scss
```
$colors: (color1: blue, color2: red, color3: green);
@each $key, $color in $colors {
.#{$color}-text {color: $color;}
}
```
“‘

Note that the $key variable is needed to reference the keys in the map. Otherwise, the compiled CSS would have color1, color2... in it.
Both of the above code examples are converted into the following CSS:
“‘scss
```
.blue-text {
color: blue;
}
.red-text {
color: red;
}
.green-text {
color: green;
}
```
“‘

3.7 Apply a Style Until a Condition is Met with @while

The @while directive is an option with similar functionality to the JavaScript while loop. It creates CSS rules until a condition is met.
The @for challenge gave an example to create a simple grid system. This can also work with @while.
“‘scss
```
$x: 1;
@while $x < 13 {
.col-#{$x} { width: 100%/12 * $x;}
$x: $x + 1;
}
```
“‘
First, define a variable $x and set it to 1. Next, use the @while directive to create the grid system while $x is less than 13.
After setting the CSS rule for width, $x is incremented by 1 to avoid an infinite loop.

3.8 Split Your Styles into Smaller Chunks with Partials

Partials in Sass are separate files that hold segments of CSS code. These are imported and used in other Sass files. This is a great way to group similar code into a module to keep it organized.
Names for partials start with the underscore (_) character, which tells Sass it is a small segment of CSS and not to convert it into a CSS file. Also, Sass files end with the .scss file extension. To bring the code in the partial into another Sass file, use the @import directive.
For example, if all your mixins are saved in a partial named "_mixins.scss", and they are needed in the "main.scss" file, this is how to use them in the main file:
“‘scss
```
// In the main.scss file
@import ’mixins’
```
“‘
Note that the underscore and file extension are not needed in the import statement - Sass understands it is a partial. Once a partial is imported into a file, all variables, mixins, and other code are available to use.

3.9 Extend One Set of CSS Styles to Another Element

Sass has a feature called extend that makes it easy to borrow the CSS rules from one element and build upon them in another.
For example, the below block of CSS rules style a .panel class. It has a background-color, height and border.
“‘scss

```
.panel{
background-color: red;
height: 70px;
border: 2px solid green;
}
```

“‘
Now you want another panel called .big-panel. It has the same base properties as .panel, but also needs a width and font-size.
It’s possible to copy and paste the initial CSS rules from .panel, but the code becomes repetitive as you add more types of panels.
The extend directive is a simple way to reuse the rules written for one element, then add more for another:
“‘scss

```
.big-panel{
@extend .panel;
width: 150px;
font-size: 2em;
}
```

“‘
The .big-panel will have the same properties as .panel in addition to the new styles.

4 React

4.1 Create a Simple JSX Element

Intro: React is an Open Source view library created and maintained by Facebook. It’s a great tool to render the User Interface (UI) of modern web applications.
React uses a syntax extension of JavaScript called JSX that allows you to write HTML directly within JavaScript. This has several benefits. It lets you use the full programmatic power of JavaScript within HTML, and helps to keep your code readable. For the most part, JSX is similar to the HTML that you have already learned, however there are a few key differences that will be covered throughout these challenges.
For instance, because JSX is a syntactic extension of JavaScript, you can actually write JavaScript directly within JSX. To do this, you simply include the code you want to be treated as JavaScript within curly braces: { ’this is treated as JavaScript code’ }. Keep this in mind, since it’s used in several future challenges.
However, because JSX is not valid JavaScript, JSX code must be compiled into JavaScript. The transpiler Babel is a popular tool for this process. For your convenience, it’s already added behind the scenes for these challenges. If you happen to write syntactically invalid JSX, you will see the first test in these challenges fail.
It’s worth noting that under the hood the challenges are calling ReactDOM.render(JSX, document.getElementById(’root’)).
This function call is what places your JSX into React’s own lightweight representation of the DOM. React then uses snapshots of its own DOM to optimize updating only specific parts of the actual DOM.

4.2 Create a Complex JSX Element

The last challenge was a simple example of JSX, but JSX can represent more complex HTML as well.
One important thing to know about nested JSX is that it must return a single element.
This one parent element would wrap all of the other levels of nested elements.
For instance, several JSX elements written as siblings with no parent wrapper element will not transpile.
Here’s an example:
Valid JSX:
“‘jsx
Paragraph One
Paragraph Two
Paragraph Three
“‘
Invalid JSX:
“‘jsx
Paragraph One
Paragraph Two
Paragraph Three
“‘

4.3 Add Comments in JSX

JSX is a syntax that gets compiled into valid JavaScript. Sometimes, for readability, you might need to add comments to your code. Like most programming languages, JSX has its own way to do this.
To put comments inside JSX, you use the syntax {/* */} to wrap around the comment text.

4.4 Render HTML Elements to the DOM

So far, you’ve learned that JSX is a convenient tool to write readable HTML within JavaScript. With React, we can render this JSX directly to the HTML DOM using React’s rendering API known as ReactDOM.
ReactDOM offers a simple method to render React elements to the DOM which looks like this: ReactDOM.render(componentToRender, targetNode), where the first argument is the React element or component

that you want to render, and the second argument is the DOM node that you want to render the component to.
As you would expect, ReactDOM.render() must be called after the JSX element declarations, just like how you must declare variables before using them.

4.5 Define an HTML Class in JSX

Now that you're getting comfortable writing JSX, you may be wondering how it differs from HTML.
So far, it may seem that HTML and JSX are exactly the same.
One key difference in JSX is that you can no longer use the word class to define HTML classes. This is because class is a reserved word in JavaScript. Instead, JSX uses className.
In fact, the naming convention for all HTML attributes and event references in JSX become camelCase. For example, a click event in JSX is onClick, instead of onclick. Likewise, onchange becomes onChange. While this is a subtle difference, it is an important one to keep in mind moving forward.

4.6 Learn About Self-Closing JSX Tags

So far, you've seen how JSX differs from HTML in a key way with the use of className vs. class for defining HTML classes.
Another important way in which JSX differs from HTML is in the idea of the self-closing tag.
In HTML, almost all tags have both an opening and closing tag: <div></div>; the closing tag always has a forward slash before the tag name that you are closing. However, there are special instances in HTML called "self-closing tags", or tags that don't require both an opening and closing tag before another tag can start.
For example the line-break tag can be written as <br> or as <br />, but should never be written as <br></br>, since it doesn't contain any content.
In JSX, the rules are a little different. Any JSX element can be written with a self-closing tag, and every element must be closed. The line-break tag, for example, must always be written as <br /> in order to be valid JSX that can be transpiled. A <div>, on the other hand, can be written as <div /> or <div></div>. The difference is that in the first syntax version there is no way to include anything in the <div />. You will see in later challenges that this syntax is useful when rendering React components.

4.7 Create a Stateless Functional Component

Components are the core of React. Everything in React is a component and here you will learn how to create one.
There are two ways to create a React component. The first way is to use a JavaScript function. Defining a component in this way creates a stateless functional component. The concept of state in an application will be covered in later challenges. For now, think of a stateless component as one that can receive data and render it, but does not manage or track changes to that data. (We'll cover the second way to create a React component in the next challenge.)
To create a component with a function, you simply write a JavaScript function that returns either JSX or null. One important thing to note is that React requires your function name to begin with a capital letter.
Here's an example of a stateless functional component that assigns an HTML class in JSX:
“‘jsx
// After being transpiled, the will have a CSS class of 'customClass'
const DemoComponent = function() {
return (

);
};
“‘
Because a JSX component represents HTML, you could put several components together to create a more

complex HTML page. This is one of the key advantages of the component architecture React provides. It allows you to compose your UI from many separate, isolated components. This makes it easier to build and maintain complex user interfaces.

4.8 Create a React Component

The other way to define a React component is with the ES6 class syntax. In the following example, Kitten extends React.Component:
“‘jsx
```
class Kitten extends React.Component {
constructor(props) {
super(props);
}
render() {
return (
Hi
);
}
}
```
“‘
This creates an ES6 class Kitten which extends the React.Component class. So the Kitten class now has access to many useful React features, such as local state and lifecycle hooks. Don’t worry if you aren’t familiar with these terms yet, they will be covered in greater detail in later challenges.
Also notice the Kitten class has a constructor defined within it that calls super(). It uses super() to call the constructor of the parent class, in this case React.Component. The constructor is a special method used during the initialization of objects that are created with the class keyword. It is best practice to call a component’s constructor with super, and pass props to both. This makes sure the component is initialized properly. For now, know that it is standard for this code to be included. Soon you will see other uses for the constructor as well as props.

4.9 Create a Component with Composition

Now we will look at how we can compose multiple React components together. Imagine you are building an App and have created three components, a Navbar, Dashboard, and Footer.
To compose these components together, you could create an App parent component which renders each of these three components as children. To render a component as a child in a React component, you include the component name written as a custom HTML tag in the JSX. For example, in the render method you could write:
“‘jsx
```
return (



)
```
“‘
When React encounters a custom HTML tag that references another component (a component name wrapped in < /> like in this example), it renders the markup for that component in the location of the tag. This should illustrate the parent/child relationship between the App component and the Navbar, Dashboard, and Footer.

4.10 Use React to Render Nested Components

The last challenge showed a simple way to compose two components, but there are many different ways you can compose components with React.
Component composition is one of React's powerful features. When you work with React, it is important to start thinking about your user interface in terms of components like the App example in the last challenge. You break down your UI into its basic building blocks, and those pieces become the components. This helps to separate the code responsible for the UI from the code responsible for handling your application logic. It can greatly simplify the development and maintenance of complex projects.

4.11 Compose React Components

As the challenges continue to use more complex compositions with React components and JSX, there is one important point to note. Rendering ES6 style class components within other components is no different than rendering the simple components you used in the last few challenges. You can render JSX elements, stateless functional components, and ES6 class components within other components.

4.12 Render a Class Component to the DOM

You may remember using the ReactDOM API in an earlier challenge to render JSX elements to the DOM. The process for rendering React components will look very similar. The past few challenges focused on components and composition, so the rendering was done for you behind the scenes. However, none of the React code you write will render to the DOM without making a call to the ReactDOM API.
Here's a refresher on the syntax: ReactDOM.render(componentToRender, targetNode). The first argument is the React component that you want to render. The second argument is the DOM node that you want to render that component within.
React components are passed into ReactDOM.render() a little differently than JSX elements. For JSX elements, you pass in the name of the element that you want to render. However, for React components, you need to use the same syntax as if you were rendering a nested component, for example ReactDOM.render(<ComponentToRender />, targetNode). You use this syntax for both ES6 class components and functional components.

4.13 Write a React Component from Scratch

Now that you've learned the basics of JSX and React components, it's time to write a component on your own. React components are the core building blocks of React applications so it's important to become very familiar with writing them. Remember, a typical React component is an ES6 class which extends React.Component. It has a render method that returns HTML (from JSX) or null. This is the basic form of a React component. Once you understand this well, you will be prepared to start building more complex React projects.

4.14 Pass Props to a Stateless Functional Component

The previous challenges covered a lot about creating and composing JSX elements, functional components, and ES6 style class components in React. With this foundation, it's time to look at another feature very common in React: props. In React, you can pass props, or properties, to child components. Say you have an App component which renders a child component called Welcome which is a stateless functional component. You can pass Welcome a user property by writing:
"'jsx
"'
You use custom HTML attributes created by you and supported by React to be passed to the component. In this case, the created property user is passed to the component Welcome. Since Welcome is a stateless

functional component, it has access to this value like so:
“‘jsx
const Welcome = (props) => Hello, {props.user}!
“‘
It is standard to call this value props and when dealing with stateless functional components, you basically consider it as an argument to a function which returns JSX. You can access the value of the argument in the function body. With class components, you will see this is a little different.

4.15 Pass an Array as Props

The last challenge demonstrated how to pass information from a parent component to a child component as props or properties. This challenge looks at how arrays can be passed as props. To pass an array to a JSX element, it must be treated as JavaScript and wrapped in curly braces.
“‘jsx
“‘
The child component then has access to the array property colors. Array methods such as join() can be used when accessing the property.
const ChildComponent = (props) => <p>{props.colors.join(', ')}</p>
This will join all colors array items into a comma separated string and produce:
<p>green, blue, red</p>
Later, we will learn about other common methods to render arrays of data in React.

4.16 Use Default Props

React also has an option to set default props. You can assign default props to a component as a property on the component itself and React assigns the default prop if necessary. This allows you to specify what a prop value should be if no value is explicitly provided. For example, if you declare MyComponent.defaultProps = { location: 'San Francisco' }, you have defined a location prop that's set to the string San Francisco, unless you specify otherwise. React assigns default props if props are undefined, but if you pass null as the value for a prop, it will remain null.

4.17 Override Default Props

The ability to set default props is a useful feature in React. The way to override the default props is to explicitly set the prop values for a component.

4.18 Use PropTypes to Define the Props You Expect

React provides useful type-checking features to verify that components receive props of the correct type. For example, your application makes an API call to retrieve data that you expect to be in an array, which is then passed to a component as a prop. You can set propTypes on your component to require the data to be of type array. This will throw a useful warning when the data is of any other type.
It's considered a best practice to set propTypes when you know the type of a prop ahead of time. You can define a propTypes property for a component in the same way you defined defaultProps. Doing this will check that props of a given key are present with a given type. Here's an example to require the type function for a prop called handleClick:
MyComponent.propTypes = { handleClick: PropTypes.func.isRequired }
In the example above, the PropTypes.func part checks that handleClick is a function. Adding isRequired tells React that handleClick is a required property for that component. You will see a warning if that prop isn't provided. Also notice that func represents function. Among the seven JavaScript primitive types, function and boolean (written as bool) are the only two that use unusual spelling. In addition to the primitive types,

there are other types available. For example, you can check that a prop is a React element. Please refer to the [documentation](https://reactjs.org/docs/jsx-in-depth.html#specifying-the-react-element-type) for all of the options.
Note: As of React v15.5.0, PropTypes is imported independently from React, like this:
import PropTypes from 'prop-types';

4.19 Access Props Using this.props

The last several challenges covered the basic ways to pass props to child components. But what if the child component that you're passing a prop to is an ES6 class component, rather than a stateless functional component? The ES6 class component uses a slightly different convention to access props.
Anytime you refer to a class component within itself, you use the this keyword. To access props within a class component, you preface the code that you use to access it with this. For example, if an ES6 class component has a prop called data, you write {this.props.data} in JSX.

4.20 Review Using Props with Stateless Functional Components

Except for the last challenge, you've been passing props to stateless functional components. These components act like pure functions. They accept props as input and return the same view every time they are passed the same props. You may be wondering what state is, and the next challenge will cover it in more detail. Before that, here's a review of the terminology for components.
A stateless functional component is any function you write which accepts props and returns JSX. A stateless component, on the other hand, is a class that extends React.Component, but does not use internal state (covered in the next challenge). Finally, a stateful component is a class component that does maintain its own internal state. You may see stateful components referred to simply as components or React components.
A common pattern is to try to minimize statefulness and to create stateless functional components wherever possible. This helps contain your state management to a specific area of your application. In turn, this improves development and maintenance of your app by making it easier to follow how changes to state affect its behavior.

4.21 Create a Stateful Component

One of the most important topics in React is state. State consists of any data your application needs to know about, that can change over time. You want your apps to respond to state changes and present an updated UI when necessary. React offers a nice solution for the state management of modern web applications.
You create state in a React component by declaring a state property on the component class in its constructor. This initializes the component with state when it is created. The state property must be set to a JavaScript object. Declaring it looks like this:
“‘jsx
this.state = {
// describe your state here
}
“‘
You have access to the state object throughout the life of your component. You can update it, render it in your UI, and pass it as props to child components. The state object can be as complex or as simple as you need it to be. Note that you must create a class component by extending React.Component in order to create state like this.

4.22 Render State in the User Interface

Once you define a component’s initial state, you can display any part of it in the UI that is rendered. If a component is stateful, it will always have access to the data in state in its render() method. You can access the data with this.state.
If you want to access a state value within the return of the render method, you have to enclose the value in curly braces.
State is one of the most powerful features of components in React. It allows you to track important data in your app and render a UI in response to changes in this data. If your data changes, your UI will change. React uses what is called a virtual DOM, to keep track of changes behind the scenes. When state data updates, it triggers a re-render of the components using that data - including child components that received the data as a prop. React updates the actual DOM, but only where necessary. This means you don’t have to worry about changing the DOM. You simply declare what the UI should look like.
Note that if you make a component stateful, no other components are aware of its state. Its state is completely encapsulated, or local to that component, unless you pass state data to a child component as props. This notion of encapsulated state is very important because it allows you to write certain logic, then have that logic contained and isolated in one place in your code.

4.23 Render State in the User Interface Another Way

There is another way to access state in a component. In the render() method, before the return statement, you can write JavaScript directly. For example, you could declare functions, access data from state or props, perform computations on this data, and so on. Then, you can assign any data to variables, which you have access to in the return statement.

4.24 Set State with this.setState

The previous challenges covered component state and how to initialize state in the constructor. There is also a way to change the component’s state. React provides a method for updating component state called setState. You call the setState method within your component class like so: this.setState(), passing in an object with key-value pairs. The keys are your state properties and the values are the updated state data. For instance, if we were storing a username in state and wanted to update it, it would look like this:
“‘jsx
this.setState({
username: ’Lewis’
});
“‘
React expects you to never modify state directly, instead always use this.setState() when state changes occur. Also, you should note that React may batch multiple state updates in order to improve performance. What this means is that state updates through the setState method can be asynchronous. There is an alternative syntax for the setState method which provides a way around this problem. This is rarely needed but it’s good to keep it in mind! Please consult the React documentation for further details.

4.25 Bind ’this’ to a Class Method

In addition to setting and updating state, you can also define methods for your component class. A class method typically needs to use the this keyword so it can access properties on the class (such as state and props) inside the scope of the method. There are a few ways to allow your class methods to access this.
One common way is to explicitly bind this in the constructor so this becomes bound to the class methods when the component is initialized. You may have noticed the last challenge used this.handleClick = this.handleClick.bind(this) for its handleClick method in the constructor. Then, when you call a function like this.setState() within your class method, this refers to the class and will not be undefined.

Note: The this keyword is one of the most confusing aspects of JavaScript but it plays an important role in React. Although its behavior here is totally normal, these lessons aren't the place for an in-depth review of this so please refer to other lessons if the above is confusing!

4.26 Use State to Toggle an Element

Sometimes you might need to know the previous state when updating the state. However, state updates may be asynchronous - this means React may batch multiple setState() calls into a single update. This means you can't rely on the previous value of this.state or this.props when calculating the next value. So, you should not use code like this:
“‘jsx
this.setState({
counter: this.state.counter + this.props.increment
});
“‘
Instead, you should pass setState a function that allows you to access state and props. Using a function with setState guarantees you are working with the most current values of state and props. This means that the above should be rewritten as:
“‘jsx
this.setState((state, props) => ({
counter: state.counter + props.increment
}));
“‘
You can also use a form without ‘props‘ if you need only the ‘state‘:
“‘jsx
this.setState(state => ({
counter: state.counter + 1
}));
“‘
Note that you have to wrap the object literal in parentheses, otherwise JavaScript thinks it's a block of code.

4.27 Write a Simple Counter

You can design a more complex stateful component by combining the concepts covered so far. These include initializing state, writing methods that set state, and assigning click handlers to trigger these methods.

4.28 Create a Controlled Input

Your application may have more complex interactions between state and the rendered UI. For example, form control elements for text input, such as input and textarea, maintain their own state in the DOM as the user types. With React, you can move this mutable state into a React component's state. The user's input becomes part of the application state, so React controls the value of that input field. Typically, if you have React components with input fields the user can type into, it will be a controlled input form.

4.29 Create a Controlled Form

The last challenge showed that React can control the internal state for certain elements like input and textarea, which makes them controlled components. This applies to other form elements as well, including the regular HTML form element.

4.30 Pass State as Props to Child Components

You saw a lot of examples that passed props to child JSX elements and child React components in previous challenges. You may be wondering where those props come from. A common pattern is to have a stateful component containing the state important to your app, that then renders child components. You want these components to have access to some pieces of that state, which are passed in as props.
For example, maybe you have an App component that renders a Navbar, among other components. In your App, you have state that contains a lot of user information, but the Navbar only needs access to the user's username so it can display it. You pass that piece of state to the Navbar component as a prop.
This pattern illustrates some important paradigms in React. The first is unidirectional data flow. State flows in one direction down the tree of your application's components, from the stateful parent component to child components. The child components only receive the state data they need. The second is that complex stateful apps can be broken down into just a few, or maybe a single, stateful component. The rest of your components simply receive state from the parent as props, and render a UI from that state. It begins to create a separation where state management is handled in one part of code and UI rendering in another. This principle of separating state logic from UI logic is one of React's key principles. When it's used correctly, it makes the design of complex, stateful applications much easier to manage.

4.31 Pass a Callback as Props

You can pass state as props to child components, but you're not limited to passing data. You can also pass handler functions or any method that's defined on a React component to a child component. This is how you allow child components to interact with their parent components. You pass methods to a child just like a regular prop. It's assigned a name and you have access to that method name under this.props in the child component.

4.32 Use the Lifecycle Method componentWillMount

React components have several special methods that provide opportunities to perform actions at specific points in the lifecycle of a component. These are called lifecycle methods, or lifecycle hooks, and allow you to catch components at certain points in time. This can be before they are rendered, before they update, before they receive props, before they unmount, and so on. Here is a list of some of the main lifecycle methods:
componentWillMount()
componentDidMount()
shouldComponentUpdate()
componentDidUpdate()
componentWillUnmount()
The next several lessons will cover some of the basic use cases for these lifecycle methods.
Note: The 'componentWillMount' Lifecycle method will be deprecated in a future version of 16.X and removed in version 17. [(Source)](https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html)

4.33 Use the Lifecycle Method componentDidMount

Most web developers, at some point, need to call an API endpoint to retrieve data. If you're working with React, it's important to know where to perform this action.
The best practice with React is to place API calls or any calls to your server in the lifecycle method componentDidMount(). This method is called after a component is mounted to the DOM. Any calls to setState() here will trigger a re-rendering of your component. When you call an API in this method, and set your state with the data that the API returns, it will automatically trigger an update once you receive the data.

4.34 Add Event Listeners

The componentDidMount() method is also the best place to attach any event listeners you need to add for specific functionality. React provides a synthetic event system which wraps the native event system present in browsers. This means that the synthetic event system behaves exactly the same regardless of the user's browser - even if the native events may behave differently between different browsers.
You've already been using some of these synthetic event handlers such as onClick(). React's synthetic event system is great to use for most interactions you'll manage on DOM elements. However, if you want to attach an event handler to the document or window objects, you have to do this directly.

4.35 Optimize Re-Renders with shouldComponentUpdate

So far, if any component receives new state or new props, it re-renders itself and all its children. This is usually okay. But React provides a lifecycle method you can call when child components receive new state or props, and declare specifically if the components should update or not. The method is shouldComponentUpdate(), and it takes nextProps and nextState as parameters.
This method is a useful way to optimize performance. For example, the default behavior is that your component re-renders when it receives new props, even if the props haven't changed. You can use shouldComponentUpdate() to prevent this by comparing the props. The method must return a boolean value that tells React whether or not to update the component. You can compare the current props (this.props) to the next props (nextProps) to determine if you need to update or not, and return true or false accordingly.

4.36 Introducing Inline Styles

There are other complex concepts that add powerful capabilities to your React code. But you may be wondering about the more simple problem of how to style those JSX elements you create in React. You likely know that it won't be exactly the same as working with HTML because of the way you apply classes to JSX elements.
If you import styles from a stylesheet, it isn't much different at all. You apply a class to your JSX element using the className attribute, and apply styles to the class in your stylesheet. Another option is to apply inline styles, which are very common in ReactJS development.
You apply inline styles to JSX elements similar to how you do it in HTML, but with a few JSX differences. Here's an example of an inline style in HTML:
<div style="color: yellow; font-size: 16px">Mellow Yellow</div>
JSX elements use the style attribute, but because of the way JSX is transpiled, you can't set the value to a string. Instead, you set it equal to a JavaScript object. Here's an example:
<div style={{color: "yellow", fontSize: 16}}>Mellow Yellow</div>
Notice how we camelCase the "fontSize" property? This is because React will not accept kebab-case keys in the style object. React will apply the correct property name for us in the HTML.

4.37 Add Inline Styles in React

You may have noticed in the last challenge that there were several other syntax differences from HTML inline styles in addition to the style attribute set to a JavaScript object. First, the names of certain CSS style properties use camel case. For example, the last challenge set the size of the font with fontSize instead of font-size. Hyphenated words like font-size are invalid syntax for JavaScript object properties, so React uses camel case. As a rule, any hyphenated style properties are written using camel case in JSX.
All property value length units (like height, width, and fontSize) are assumed to be in px unless otherwise specified. If you want to use em, for example, you wrap the value and the units in quotes, like {fontSize: "4em"}. Other than the length values that default to px, all other property values should be wrapped in quotes.

4.38 Use Advanced JavaScript in React Render Method

In previous challenges, you learned how to inject JavaScript code into JSX code using curly braces, { }, for tasks like accessing props, passing props, accessing state, inserting comments into your code, and most recently, styling your components. These are all common use cases to put JavaScript in JSX, but they aren't the only way that you can utilize JavaScript code in your React components.
You can also write JavaScript directly in your render methods, before the return statement, without inserting it inside of curly braces. This is because it is not yet within the JSX code. When you want to use a variable later in the JSX code inside the return statement, you place the variable name inside curly braces.

4.39 Render with an If-Else Condition

Another application of using JavaScript to control your rendered view is to tie the elements that are rendered to a condition. When the condition is true, one view renders. When it's false, it's a different view. You can do this with a standard if/else statement in the render() method of a React component.

4.40 Use && for a More Concise Conditional

The if/else statements worked in the last challenge, but there's a more concise way to achieve the same result. Imagine that you are tracking several conditions in a component and you want different elements to render depending on each of these conditions. If you write a lot of else if statements to return slightly different UIs, you may repeat code which leaves room for error. Instead, you can use the && logical operator to perform conditional logic in a more concise way. This is possible because you want to check if a condition is true, and if it is, return some markup. Here's an example:
{condition && <p>markup</p>}
If the condition is true, the markup will be returned. If the condition is false, the operation will immediately return false after evaluating the condition and return nothing. You can include these statements directly in your JSX and string multiple conditions together by writing && after each one. This allows you to handle more complex conditional logic in your render() method without repeating a lot of code.

4.41 Use a Ternary Expression for Conditional Rendering

Before moving on to dynamic rendering techniques, there's one last way to use built-in JavaScript conditionals to render what you want: the ternary operator. The ternary operator is often utilized as a shortcut for if/else statements in JavaScript. They're not quite as robust as traditional if/else statements, but they are very popular among React developers. One reason for this is because of how JSX is compiled, if/else statements can't be inserted directly into JSX code. You might have noticed this a couple challenges ago — when an if/else statement was required, it was always outside the return statement. Ternary expressions can be an excellent alternative if you want to implement conditional logic within your JSX. Recall that a ternary operator has three parts, but you can combine several ternary expressions together. Here's the basic syntax:
“‘jsx
condition ? expressionIfTrue : expressionIfFalse;
“‘

4.42 Render Conditionally from Props

So far, you've seen how to use if/else, &&, null and the ternary operator (condition ? expressionIfTrue : expressionIfFalse) to make conditional decisions about what to render and when. However, there's one important topic left to discuss that lets you combine any or all of these concepts with another powerful React feature: props. Using props to conditionally render code is very common with React developers — that is, they use the value of a given prop to automatically make decisions about what to render.

In this challenge, you'll set up a child component to make rendering decisions based on props. You'll also use the ternary operator, but you can see how several of the other concepts that were covered in the last few challenges might be just as useful in this context.

4.43 Change Inline CSS Conditionally Based on Component State

At this point, you've seen several applications of conditional rendering and the use of inline styles. Here's one more example that combines both of these topics. You can also render CSS conditionally based on the state of a React component. To do this, you check for a condition, and if that condition is met, you modify the styles object that's assigned to the JSX elements in the render method.
This paradigm is important to understand because it is a dramatic shift from the more traditional approach of applying styles by modifying DOM elements directly (which is very common with jQuery, for example). In that approach, you must keep track of when elements change and also handle the actual manipulation directly. It can become difficult to keep track of changes, potentially making your UI unpredictable. When you set a style object based on a condition, you describe how the UI should look as a function of the application's state. There is a clear flow of information that only moves in one direction. This is the preferred method when writing applications with React.

4.44 Use Array.map() to Dynamically Render Elements

Conditional rendering is useful, but you may need your components to render an unknown number of elements. Often in reactive programming, a programmer has no way to know what the state of an application is until runtime, because so much depends on a user's interaction with that program. Programmers need to write their code to correctly handle that unknown state ahead of time. Using Array.map() in React illustrates this concept.
For example, you create a simple "To Do List" app. As the programmer, you have no way of knowing how many items a user might have on their list. You need to set up your component to dynamically render the correct number of list elements long before someone using the program decides that today is laundry day.

4.45 Give Sibling Elements a Unique Key Attribute

The last challenge showed how the map method is used to dynamically render a number of elements based on user input. However, there was an important piece missing from that example. When you create an array of elements, each one needs a key attribute set to a unique value. React uses these keys to keep track of which items are added, changed, or removed. This helps make the re-rendering process more efficient when the list is modified in any way.Note: Keys only need to be unique between sibling elements, they don't need to be globally unique in your application.

4.46 Use Array.filter() to Dynamically Filter an Array

The map array method is a powerful tool that you will use often when working with React. Another method related to map is filter, which filters the contents of an array based on a condition, then returns a new array. For example, if you have an array of users that all have a property online which can be set to true or false, you can filter only those users that are online by writing:
let onlineUsers = users.filter(user => user.online);

4.47 Render React on the Server with renderToString

So far, you have been rendering React components on the client. Normally, this is what you will always do. However, there are some use cases where it makes sense to render a React component on the server. Since

React is a JavaScript view library and you can run JavaScript on the server with Node, this is possible. In fact, React provides a renderToString() method you can use for this purpose.
There are two key reasons why rendering on the server may be used in a real world app. First, without doing this, your React apps would consist of a relatively empty HTML file and a large bundle of JavaScript when it's initially loaded to the browser. This may not be ideal for search engines that are trying to index the content of your pages so people can find you. If you render the initial HTML markup on the server and send this to the client, the initial page load contains all of the page's markup which can be crawled by search engines. Second, this creates a faster initial page load experience because the rendered HTML is smaller than the JavaScript code of the entire app. React will still be able to recognize your app and manage it after the initial load.

5 Redux

5.1 Create a Redux Store

Redux is a state management framework that can be used with a number of different web technologies, including React.
In Redux, there is a single state object that's responsible for the entire state of your application. This means if you had a React app with ten components, and each component had its own local state, the entire state of your app would be defined by a single state object housed in the Redux store. This is the first important principle to understand when learning Redux: the Redux store is the single source of truth when it comes to application state.
This also means that any time any piece of your app wants to update state, it must do so through the Redux store. The unidirectional data flow makes it easier to track state management in your app.

5.2 Get State from the Redux Store

The Redux store object provides several methods that allow you to interact with it. For example, you can retrieve the current state held in the Redux store object with the getState() method.

5.3 Define a Redux Action

Since Redux is a state management framework, updating state is one of its core tasks. In Redux, all state updates are triggered by dispatching actions. An action is simply a JavaScript object that contains information about an action event that has occurred. The Redux store receives these action objects, then updates its state accordingly. Sometimes a Redux action also carries some data. For example, the action carries a username after a user logs in. While the data is optional, actions must carry a type property that specifies the 'type' of action that occurred.
Think of Redux actions as messengers that deliver information about events happening in your app to the Redux store. The store then conducts the business of updating state based on the action that occurred.

5.4 Define an Action Creator

After creating an action, the next step is sending the action to the Redux store so it can update its state. In Redux, you define action creators to accomplish this. An action creator is simply a JavaScript function that returns an action. In other words, action creators create objects that represent action events.

5.5 Dispatch an Action Event

dispatch method is what you use to dispatch actions to the Redux store. Calling store.dispatch() and passing the value returned from an action creator sends an action back to the store.
Recall that action creators return an object with a type property that specifies the action that has occurred. Then the method dispatches an action object to the Redux store. Based on the previous challenge's example, the following lines are equivalent, and both dispatch the action of type LOGIN:
“‘js
store.dispatch(actionCreator());
store.dispatch({ type: 'LOGIN' });
“‘

5.6 Handle an Action in the Store

After an action is created and dispatched, the Redux store needs to know how to respond to that action. This is the job of a reducer function. Reducers in Redux are responsible for the state modifications that take place in response to actions. A reducer takes state and action as arguments, and it always returns a new state. It is important to see that this is the only role of the reducer. It has no side effects — it never calls an API endpoint and it never has any hidden surprises. The reducer is simply a pure function that takes state and action, then returns new state.
Another key principle in Redux is that state is read-only. In other words, the reducer function must always return a new copy of state and never modify state directly. Redux does not enforce state immutability, however, you are responsible for enforcing it in the code of your reducer functions. You'll practice this in later challenges.

5.7 Use a Switch Statement to Handle Multiple Actions

You can tell the Redux store how to handle multiple action types. Say you are managing user authentication in your Redux store. You want to have a state representation for when users are logged in and when they are logged out. You represent this with a single state object with the property authenticated. You also need action creators that create actions corresponding to user login and user logout, along with the action objects themselves.

5.8 Use const for Action Types

A common practice when working with Redux is to assign action types as read-only constants, then reference these constants wherever they are used. You can refactor the code you're working with to write the action types as const declarations.

5.9 Register a Store Listener

Another method you have access to on the Redux store object is store.subscribe(). This allows you to subscribe listener functions to the store, which are called whenever an action is dispatched against the store. One simple use for this method is to subscribe a function to your store that simply logs a message every time an action is received and the store is updated.

5.10 Combine Multiple Reducers

When the state of your app begins to grow more complex, it may be tempting to divide state into multiple pieces. Instead, remember the first principle of Redux: all app state is held in a single state object in the store. Therefore, Redux provides reducer composition as a solution for a complex state model. You define multiple reducers to handle different pieces of your application's state, then compose these reducers together into one root reducer. The root reducer is then passed into the Redux createStore() method.
In order to let us combine multiple reducers together, Redux provides the combineReducers() method. This method accepts an object as an argument in which you define properties which associate keys to specific reducer functions. The name you give to the keys will be used by Redux as the name for the associated piece of state.
Typically, it is a good practice to create a reducer for each piece of application state when they are distinct or unique in some way. For example, in a note-taking app with user authentication, one reducer could handle authentication while another handles the text and notes that the user is submitting. For such an application, we might write the combineReducers() method like this:
“‘js
const rootReducer = Redux.combineReducers({

```
auth: authenticationReducer,
notes: notesReducer
});
```
“‘
Now, the key notes will contain all of the state associated with our notes and handled by our notesReducer. This is how multiple reducers can be composed to manage more complex application state. In this example, the state held in the Redux store would then be a single object containing auth and notes properties.

5.11 Send Action Data to the Store

By now you’ve learned how to dispatch actions to the Redux store, but so far these actions have not contained any information other than a type. You can also send specific data along with your actions. In fact, this is very common because actions usually originate from some user interaction and tend to carry some data with them. The Redux store often needs to know about this data.

5.12 Use Middleware to Handle Asynchronous Actions

So far these challenges have avoided discussing asynchronous actions, but they are an unavoidable part of web development. At some point you’ll need to call asynchronous endpoints in your Redux app, so how do you handle these types of requests? Redux provides middleware designed specifically for this purpose, called Redux Thunk middleware. Here’s a brief description how to use this with Redux.
To include Redux Thunk middleware, you pass it as an argument to Redux.applyMiddleware(). This statement is then provided as a second optional parameter to the createStore() function. Take a look at the code at the bottom of the editor to see this. Then, to create an asynchronous action, you return a function in the action creator that takes dispatch as an argument. Within this function, you can dispatch actions and perform asynchronous requests.
In this example, an asynchronous request is simulated with a setTimeout() call. It’s common to dispatch an action before initiating any asynchronous behavior so that your application state knows that some data is being requested (this state could display a loading icon, for instance). Then, once you receive the data, you dispatch another action which carries the data as a payload along with information that the action is completed.
Remember that you’re passing dispatch as a parameter to this special action creator. This is what you’ll use to dispatch your actions, you simply pass the action directly to dispatch and the middleware takes care of the rest.

5.13 Write a Counter with Redux

Now you’ve learned all the core principles of Redux! You’ve seen how to create actions and action creators, create a Redux store, dispatch your actions against the store, and design state updates with pure reducers. You’ve even seen how to manage complex state with reducer composition and handle asynchronous actions. These examples are simplistic, but these concepts are the core principles of Redux. If you understand them well, you’re ready to start building your own Redux app. The next challenges cover some of the details regarding state immutability, but first, here’s a review of everything you’ve learned so far.

5.14 Never Mutate State

These final challenges describe several methods of enforcing the key principle of state immutability in Redux. Immutable state means that you never modify state directly, instead, you return a new copy of state.
If you took a snapshot of the state of a Redux app over time, you would see something like state 1, state 2, state 3,state 4, ... and so on where each state may be similar to the last, but each is a distinct piece of data. This immutability, in fact, is what provides such features as time-travel debugging that you may have heard

about.
Redux does not actively enforce state immutability in its store or reducers, that responsibility falls on the programmer. Fortunately, JavaScript (especially ES6) provides several useful tools you can use to enforce the immutability of your state, whether it is a string, number, array, or object. Note that strings and numbers are primitive values and are immutable by nature. In other words, 3 is always 3. You cannot change the value of the number 3. An array or object, however, is mutable. In practice, your state will probably consist of an array or object, as these are useful data structures for representing many types of information.

5.15 Use the Spread Operator on Arrays

One solution from ES6 to help enforce state immutability in Redux is the spread operator: .... The spread operator has a variety of applications, one of which is well-suited to the previous challenge of producing a new array from an existing array. This is relatively new, but commonly used syntax. For example, if you have an array myArray and write:
let newArray = [...myArray];
newArray is now a clone of myArray. Both arrays still exist separately in memory. If you perform a mutation like newArray.push(5), myArray doesn't change. The ... effectively spreads out the values in myArray into a new array. To clone an array but add additional values in the new array, you could write [...myArray, 'new value']. This would return a new array composed of the values in myArray and the string 'new value' as the last value. The spread syntax can be used multiple times in array composition like this, but it's important to note that it only makes a shallow copy of the array. That is to say, it only provides immutable array operations for one-dimensional arrays.

5.16 Remove an Item from an Array

Time to practice removing items from an array. The spread operator can be used here as well. Other useful JavaScript methods include slice() and concat().

5.17 Copy an Object with Object.assign

The last several challenges worked with arrays, but there are ways to help enforce state immutability when state is an object, too. A useful tool for handling objects is the Object.assign() utility. Object.assign() takes a target object and source objects and maps properties from the source objects to the target object. Any matching properties are overwritten by properties in the source objects. This behavior is commonly used to make shallow copies of objects by passing an empty object as the first argument followed by the object(s) you want to copy. Here's an example:
const newObject = Object.assign({}, obj1, obj2);
This creates newObject as a new object, which contains the properties that currently exist in obj1 and obj2.

6 React And Redux

6.1 Getting Started with React Redux

This series of challenges introduces how to use Redux with React. First, here's a review of some of the key principles of each technology. React is a view library that you provide with data, then it renders the view in an efficient, predictable way. Redux is a state management framework that you can use to simplify the management of your application's state. Typically, in a React Redux app, you create a single Redux store that manages the state of your entire app. Your React components subscribe to only the pieces of data in the store that are relevant to their role. Then, you dispatch actions directly from React components, which then trigger store updates.
Although React components can manage their own state locally, when you have a complex app, it's generally better to keep the app state in a single location with Redux. There are exceptions when individual components may have local state specific only to them. Finally, because Redux is not designed to work with React out of the box, you need to use the react-redux package. It provides a way for you to pass Redux state and dispatch to your React components as props.
Over the next few challenges, first, you'll create a simple React component which allows you to input new text messages. These are added to an array that's displayed in the view. This should be a nice review of what you learned in the React lessons. Next, you'll create a Redux store and actions that manage the state of the messages array. Finally, you'll use react-redux to connect the Redux store with your component, thereby extracting the local state into the Redux store.

6.2 Manage State Locally First

Here you'll finish creating the DisplayMessages component.

6.3 Extract State Logic to Redux

Now that you finished the React component, you need to move the logic it's performing locally in its state into Redux. This is the first step to connect the simple React app to Redux. The only functionality your app has is to add new messages from the user to an unordered list. The example is simple in order to demonstrate how React and Redux work together.

6.4 Use Provider to Connect Redux to React

In the last challenge, you created a Redux store to handle the messages array and created an action for adding new messages. The next step is to provide React access to the Redux store and the actions it needs to dispatch updates. React Redux provides its react-redux package to help accomplish these tasks.
React Redux provides a small API with two key features: Provider and connect. Another challenge covers connect. The Provider is a wrapper component from React Redux that wraps your React app. This wrapper then allows you to access the Redux store and dispatch functions throughout your component tree. Provider takes two props, the Redux store and the child components of your app. Defining the Provider for an App component might look like this:
“‘jsx
“‘

6.5 Map State to Props

The Provider component allows you to provide state and dispatch to your React components, but you must specify exactly what state and actions you want. This way, you make sure that each component only has

access to the state it needs. You accomplish this by creating two functions: mapStateToProps() and mapDispatchToProps().
In these functions, you declare what pieces of state you want to have access to and which action creators you need to be able to dispatch. Once these functions are in place, you'll see how to use the React Redux connect method to connect them to your components in another challenge.
Note: Behind the scenes, React Redux uses the store.subscribe() method to implement mapStateToProps().

6.6 Map Dispatch to Props

The mapDispatchToProps() function is used to provide specific action creators to your React components so they can dispatch actions against the Redux store. It's similar in structure to the mapStateToProps() function you wrote in the last challenge. It returns an object that maps dispatch actions to property names, which become component props. However, instead of returning a piece of state, each property returns a function that calls dispatch with an action creator and any relevant action data. You have access to this dispatch because it's passed in to mapDispatchToProps() as a parameter when you define the function, just like you passed state to mapStateToProps(). Behind the scenes, React Redux is using Redux's store.dispatch() to conduct these dispatches with mapDispatchToProps(). This is similar to how it uses store.subscribe() for components that are mapped to state.
For example, you have a loginUser() action creator that takes a username as an action payload. The object returned from mapDispatchToProps() for this action creator would look something like:
“‘jsx
{
submitLoginUser: function(username) {
dispatch(loginUser(username));
}
}
“‘

6.7 Connect Redux to React

Now that you've written both the mapStateToProps() and the mapDispatchToProps() functions, you can use them to map state and dispatch to the props of one of your React components. The connect method from React Redux can handle this task. This method takes two optional arguments, mapStateToProps() and mapDispatchToProps(). They are optional because you may have a component that only needs access to state but doesn't need to dispatch any actions, or vice versa.
To use this method, pass in the functions as arguments, and immediately call the result with your component. This syntax is a little unusual and looks like:
connect(mapStateToProps, mapDispatchToProps)(MyComponent)
Note: If you want to omit one of the arguments to the connect method, you pass null in its place.

6.8 Connect Redux to the Messages App

Now that you understand how to use connect to connect React to Redux, you can apply what you've learned to your React component that handles messages.
In the last lesson, the component you connected to Redux was named Presentational, and this wasn't arbitrary. This term generally refers to React components that are not directly connected to Redux. They are simply responsible for the presentation of UI and do this as a function of the props they receive. By contrast, container components are connected to Redux. These are typically responsible for dispatching actions to the store and often pass store state to child components as props.

6.9 Extract Local State into Redux

You're almost done! Recall that you wrote all the Redux code so that Redux could control the state management of your React messages app. Now that Redux is connected, you need to extract the state management out of the Presentational component and into Redux. Currently, you have Redux connected, but you are handling the state locally within the Presentational component.

6.10 Moving Forward From Here

Congratulations! You finished the lessons on React and Redux. There's one last item worth pointing out before you move on. Typically, you won't write React apps in a code editor like this. This challenge gives you a glimpse of what the syntax looks like if you're working with npm and a file system on your own machine. The code should look similar, except for the use of import statements (these pull in all of the dependencies that have been provided for you in the challenges). The "Managing Packages with npm" section covers npm in more detail.
Finally, writing React and Redux code generally requires some configuration. This can get complicated quickly. If you are interested in experimenting on your own machine, the
Create React App comes configured and ready to go.
Alternatively, you can enable Babel as a JavaScript Preprocessor in CodePen, add React and ReactDOM as external JavaScript resources, and work there as well.

7 Front End Libraries Projects

7.1 Build a Random Quote Machine

Objective: Build a CodePen.io app that is functionally similar to this: https://codepen.io/freeCodeCamp/full/qRZeGZ.
Fulfill the below user stories and get all of the tests to pass. Give it your own personal style.
You can use any mix of HTML, JavaScript, CSS, Bootstrap, SASS, React, Redux, and jQuery to complete this project. You should use a frontend framework (like React for example) because this section is about learning frontend frameworks. Additional technologies not listed above are not recommended and using them is at your own risk. We are looking at supporting other frontend frameworks like Angular and Vue, but they are not currently supported. We will accept and try to fix all issue reports that use the suggested technology stack for this project. Happy coding!
User Story #1: I can see a wrapper element with a corresponding id="quote-box".
User Story #2: Within #quote-box, I can see an element with a corresponding id="text".
User Story #3: Within #quote-box, I can see an element with a corresponding id="author".
User Story #4: Within #quote-box, I can see a clickable element with a corresponding id="new-quote".
User Story #5: Within #quote-box, I can see a clickable a element with a corresponding id="tweet-quote".
User Story #6: On first load, my quote machine displays a random quote in the element with id="text".
User Story #7: On first load, my quote machine displays the random quote's author in the element with id="author".
User Story #8: When the #new-quote button is clicked, my quote machine should fetch a new quote and display it in the #text element.
User Story #9: My quote machine should fetch the new quote's author when the #new-quote button is clicked and display it in the #author element.
User Story #10: I can tweet the current quote by clicking on the #tweet-quotea element. This a element should include the "twitter.com/intent/tweet" path in its href attribute to tweet the current quote.
User Story #11: The #quote-box wrapper element should be horizontally centered. Please run tests with browser's zoom level at 100% and page maximized.
You can build your project by forking this CodePen pen. Or you can use this CDN link to run the tests in any environment you like: https://cdn.freecodecamp.org/testable-projects-fcc/v1/bundle.js
Once you're done, submit the URL to your working project with all its tests passing.

7.2 Build a Markdown Previewer

Objective: Build a CodePen.io app that is functionally similar to this: https://codepen.io/freeCodeCamp/full/GrZVVO.
Fulfill the below user stories and get all of the tests to pass. Give it your own personal style.
You can use any mix of HTML, JavaScript, CSS, Bootstrap, SASS, React, Redux, and jQuery to complete this project. You should use a frontend framework (like React for example) because this section is about learning frontend frameworks. Additional technologies not listed above are not recommended and using them is at your own risk. We are looking at supporting other frontend frameworks like Angular and Vue, but they are not currently supported. We will accept and try to fix all issue reports that use the suggested technology stack for this project. Happy coding!
User Story #1: I can see a textarea element with a corresponding id="editor".
User Story #2: I can see an element with a corresponding id="preview".
User Story #3: When I enter text into the #editor element, the #preview element is updated as I type to display the content of the textarea.
User Story #4: When I enter GitHub flavored markdown into the #editor element, the text is rendered as HTML in the #preview element as I type (HINT: You don't need to parse Markdown yourself - you can import the Marked library for this: https://cdnjs.com/libraries/marked).
User Story #5: When my markdown previewer first loads, the default text in the #editor field should contain valid markdown that represents at least one of each of the following elements: a header (H1 size), a sub header (H2 size), a link, inline code, a code block, a list item, a blockquote, an image, and bolded text.
User Story #6: When my markdown previewer first loads, the default markdown in the #editor field should be rendered as HTML in the #preview element.

Optional Bonus (you do not need to make this test pass): My markdown previewer interprets carriage returns and renders them as br (line break) elements.
You can build your project by forking this CodePen pen. Or you can use this CDN link to run the tests in any environment you like: https://cdn.freecodecamp.org/testable-projects-fcc/v1/bundle.js
Once you're done, submit the URL to your working project with all its tests passing.

7.3 Build a Drum Machine

Objective: Build a CodePen.io app that is functionally similar to this: https://codepen.io/freeCodeCamp/full/MJyNMd. Fulfill the below user stories and get all of the tests to pass. Give it your own personal style.
You can use any mix of HTML, JavaScript, CSS, Bootstrap, SASS, React, Redux, and jQuery to complete this project. You should use a frontend framework (like React for example) because this section is about learning frontend frameworks. Additional technologies not listed above are not recommended and using them is at your own risk. We are looking at supporting other frontend frameworks like Angular and Vue, but they are not currently supported. We will accept and try to fix all issue reports that use the suggested technology stack for this project. Happy coding!
User Story #1: I should be able to see an outer container with a corresponding id="drum-machine" that contains all other elements.
User Story #2: Within #drum-machine I can see an element with a corresponding id="display".
User Story #3: Within #drum-machine I can see 9 clickable drum pad elements, each with a class name of drum-pad, a unique id that describes the audio clip the drum pad will be set up to trigger, and an inner text that corresponds to one of the following keys on the keyboard: Q, W, E, A, S, D, Z, X, C. The drum pads MUST be in this order.
User Story #4: Within each .drum-pad, there should be an HTML5 audio element which has a src attribute pointing to an audio clip, a class name of clip, and an id corresponding to the inner text of its parent .drum-pad (e.g. id="Q", id="W", id="E" etc.).
User Story #5: When I click on a .drum-pad element, the audio clip contained in its child audio element should be triggered.
User Story #6: When I press the trigger key associated with each .drum-pad, the audio clip contained in its child audio element should be triggered (e.g. pressing the Q key should trigger the drum pad which contains the string "Q", pressing the W key should trigger the drum pad which contains the string "W", etc.).
User Story #7: When a .drum-pad is triggered, a string describing the associated audio clip is displayed as the inner text of the #display element (each string must be unique).
You can build your project by forking this CodePen pen. Or you can use this CDN link to run the tests in any environment you like: https://cdn.freecodecamp.org/testable-projects-fcc/v1/bundle.js
Once you're done, submit the URL to your working project with all its tests passing.

7.4 Build a JavaScript Calculator

Objective: Build a CodePen.io app that is functionally similar to this: https://codepen.io/freeCodeCamp/full/wgGVVX. Fulfill the below user stories and get all of the tests to pass. Give it your own personal style.
You can use any mix of HTML, JavaScript, CSS, Bootstrap, SASS, React, Redux, and jQuery to complete this project. You should use a frontend framework (like React for example) because this section is about learning frontend frameworks. Additional technologies not listed above are not recommended and using them is at your own risk. We are looking at supporting other frontend frameworks like Angular and Vue, but they are not currently supported. We will accept and try to fix all issue reports that use the suggested technology stack for this project. Happy coding!
User Story #1: My calculator should contain a clickable element containing an = (equal sign) with a corresponding id="equals".
User Story #2: My calculator should contain 10 clickable elements containing one number each from 0-9, with the following corresponding IDs: id="zero", id="one", id="two", id="three", id="four", id="five", id="six", id="seven", id="eight", and id="nine".

User Story #3: My calculator should contain 4 clickable elements each containing one of the 4 primary mathematical operators with the following corresponding IDs: id="add", id="subtract", id="multiply", id="divide".
User Story #4: My calculator should contain a clickable element containing a . (decimal point) symbol with a corresponding id="decimal".
User Story #5: My calculator should contain a clickable element with an id="clear".
User Story #6: My calculator should contain an element to display values with a corresponding id="display".
User Story #7: At any time, pressing the clear button clears the input and output values, and returns the calculator to its initialized state; 0 should be shown in the element with the id of display.
User Story #8: As I input numbers, I should be able to see my input in the element with the id of display.
User Story #9: In any order, I should be able to add, subtract, multiply and divide a chain of numbers of any length, and when I hit =, the correct result should be shown in the element with the id of display.
User Story #10: When inputting numbers, my calculator should not allow a number to begin with multiple zeros.
User Story #11: When the decimal element is clicked, a . should append to the currently displayed value; two . in one number should not be accepted.
User Story #12: I should be able to perform any operation (+, -, *, /) on numbers containing decimal points.
User Story #13: If 2 or more operators are entered consecutively, the operation performed should be the last operator entered (excluding the negative (-) sign). For example, if 5 + * 7 = is entered, the result should be 35 (i.e. 5 * 7); if 5 * - 5 = is entered, the result should be -25 (i.e. 5 x (-5)).
User Story #14: Pressing an operator immediately following = should start a new calculation that operates on the result of the previous evaluation.
User Story #15: My calculator should have several decimal places of precision when it comes to rounding (note that there is no exact standard, but you should be able to handle calculations like 2 / 7 with reasonable precision to at least 4 decimal places).
Note On Calculator Logic: It should be noted that there are two main schools of thought on calculator input logic: immediate execution logic and formula logic. Our example utilizes formula logic and observes order of operation precedence, immediate execution does not. Either is acceptable, but please note that depending on which you choose, your calculator may yield different results than ours for certain equations (see below example). As long as your math can be verified by another production calculator, please do not consider this a bug.
EXAMPLE: 3 + 5 x 6 - 2 / 4 =Immediate Execution Logic: 11.5Formula/Expression Logic: 32.5
You can build your project by forking this CodePen pen. Or you can use this CDN link to run the tests in any environment you like: https://cdn.freecodecamp.org/testable-projects-fcc/v1/bundle.js
Once you're done, submit the URL to your working project with all its tests passing.

7.5 Build a 25 + 5 Clock

Objective: Build a CodePen.io app that is functionally similar to this: https://codepen.io/freeCodeCamp/full/XpKrrW. Fulfill the below user stories and get all of the tests to pass. Give it your own personal style.
You can use any mix of HTML, JavaScript, CSS, Bootstrap, SASS, React, Redux, and jQuery to complete this project. You should use a frontend framework (like React for example) because this section is about learning frontend frameworks. Additional technologies not listed above are not recommended and using them is at your own risk. We are looking at supporting other frontend frameworks like Angular and Vue, but they are not currently supported. We will accept and try to fix all issue reports that use the suggested technology stack for this project. Happy coding!
User Story #1: I can see an element with id="break-label" that contains a string (e.g. "Break Length").
User Story #2: I can see an element with id="session-label" that contains a string (e.g. "Session Length").
User Story #3: I can see two clickable elements with corresponding IDs: id="break-decrement" and id="session-decrement".
User Story #4: I can see two clickable elements with corresponding IDs: id="break-increment" and id="session-increment".
User Story #5: I can see an element with a corresponding id="break-length", which by default (on load) displays a value of 5.

User Story #6: I can see an element with a corresponding id="session-length", which by default displays a value of 25.
User Story #7: I can see an element with a corresponding id="timer-label", that contains a string indicating a session is initialized (e.g. "Session").
User Story #8: I can see an element with corresponding id="time-left". NOTE: Paused or running, the value in this field should always be displayed in mm:ss format (i.e. 25:00).
User Story #9: I can see a clickable element with a corresponding id="start_stop".
User Story #10: I can see a clickable element with a corresponding id="reset".
User Story #11: When I click the element with the id of reset, any running timer should be stopped, the value within id="break-length" should return to 5, the value within id="session-length" should return to 25, and the element with id="time-left" should reset to it's default state.
User Story #12: When I click the element with the id of break-decrement, the value within id="break-length" decrements by a value of 1, and I can see the updated value.
User Story #13: When I click the element with the id of break-increment, the value within id="break-length" increments by a value of 1, and I can see the updated value.
User Story #14: When I click the element with the id of session-decrement, the value within id="session-length" decrements by a value of 1, and I can see the updated value.
User Story #15: When I click the element with the id of session-increment, the value within id="session-length" increments by a value of 1, and I can see the updated value.
User Story #16: I should not be able to set a session or break length to <= 0.
User Story #17: I should not be able to set a session or break length to > 60.
User Story #18: When I first click the element with id="start_stop", the timer should begin running from the value currently displayed in id="session-length", even if the value has been incremented or decremented from the original value of 25.
User Story #19: If the timer is running, the element with the id of time-left should display the remaining time in mm:ss format (decrementing by a value of 1 and updating the display every 1000ms).
User Story #20: If the timer is running and I click the element with id="start_stop", the countdown should pause.
User Story #21: If the timer is paused and I click the element with id="start_stop", the countdown should resume running from the point at which it was paused.
User Story #22: When a session countdown reaches zero (NOTE: timer MUST reach 00:00), and a new countdown begins, the element with the id of timer-label should display a string indicating a break has begun.
User Story #23: When a session countdown reaches zero (NOTE: timer MUST reach 00:00), a new break countdown should begin, counting down from the value currently displayed in the id="break-length" element.
User Story #24: When a break countdown reaches zero (NOTE: timer MUST reach 00:00), and a new countdown begins, the element with the id of timer-label should display a string indicating a session has begun.
User Story #25: When a break countdown reaches zero (NOTE: timer MUST reach 00:00), a new session countdown should begin, counting down from the value currently displayed in the id="session-length" element.
User Story #26: When a countdown reaches zero (NOTE: timer MUST reach 00:00), a sound indicating that time is up should play. This should utilize an HTML5 audio tag and have a corresponding id="beep".
User Story #27: The audio element with id="beep" must be 1 second or longer.
User Story #28: The audio element with id of beep must stop playing and be rewound to the beginning when the element with the id of reset is clicked.
You can build your project by forking this CodePen pen. Or you can use this CDN link to run the tests in any environment you like: https://cdn.freecodecamp.org/testable-projects-fcc/v1/bundle.js
Once you're done, submit the URL to your working project with all its tests passing.