PERSEVERE

Student Guide to Coding Front End Libraries

Introduction to Bootstrap

Bootstrap is a front-end framework used to design responsive web pages and web applications. It takes a mobile-first approach to web development. Bootstrap includes pre-built CSS styles and classes, plus some JavaScript functionality. Bootstrap uses a responsive 12 column grid layout and has design templates for:

- buttons
- images
- tables
- forms
- navigation

Bootstrap is an open source toolkit for developing with HTML, CSS, and JS. Quickly prototype your ideas or build your entire app with our Sass variables and mixins, responsive grid system, extensive pre-built components, and powerful plugins built on jQuery.

Bootstrap Classes

container-fluid	This class makes sure that the container will adjust to the screen size of the device browsing the website.
img-responsive	This class will make sure the image will adjust to the browser of the device viewing the website.
text-center	centers the text
btn btn-default	Gives a button the default bootstrap button style.
btn-block	Makes the button take up the whole width.
btn btn-primary	Gives a button the primary bootstrap button style. Adds Color
btn-info	Uses a color that calls attention to optional actions.
btn-danger	Uses a button color that will notify users that the button performs a destructive action.

text-primary	Gives the text the primary bootstrap color.
text-danger	Gives the text a red color
form-control	All <input>,<textarea>, and <select> elements with the class form-control have a width of 100%
well	Creates a visual sense of depth for your columns.

Bootstrap Grid

Bootstrap uses a responsive 12-column grid system, which makes it easy to put elements into rows and specify each element's relative width. Many of the classes can be applied to a div element.

```
<div class="container">
  <div class="row">
    <div class="col-sm-4">
      One of three columns
    </div>
    <div class="col-sm-4">
      One of three columns
    </div>
    <div class="col-sm-4">
      One of three columns
    </div>
  </div>
</div>
```

All bootstrap rows should equal up to 12-columns, and all rows should have a container.

col-xs-*	This is for an area less than 576 px and the * represents how many columns wide the object will be
col-s-*	This is for an area between 540 to 720 px

	and the * represents how many columns wide the object will be
col-md-*	This is for an area between 720 to 960 px and the * represents how many columns wide the object will be
col-lg-*	This for an area between 960 to 1140 px and the * represents how many columns wide the object will be.
col-xl-*	This is for an area larger than 1140 px and the * represents how many columns wide the object will be.

Font Awesome Icons

Font Awesome is a library of icons. The icons are vector graphics, stored in a .svg file format. These icons are treated just like fonts. You can specify size and they will assume the font size of their parent HTML elements.

```
<!-- The i element, used in the past for italics. Now it is used for icons. We can also use the span element -->

<i class="fa fat-info-circle"> </i>
```

fa fa-info-circle	A class that puts an info icon onto the website.
fa fa-thumbs-up	A class that puts a thumbs up into the website.
fa fa-trash	A class that places a trash can icon into the website.
fa fa-paper-plane	Input's a paper plane into the website.

Basic Bootstrap Review Questions

1. What is Bootstrap?

2. How many column grid layout does Bootstrap use?

3. What attribute do you use to apply Bootstrap?

4. What class places a trash can icon into the website?

5. What class will make sure the image will adjust to the browser of the device viewing the website?

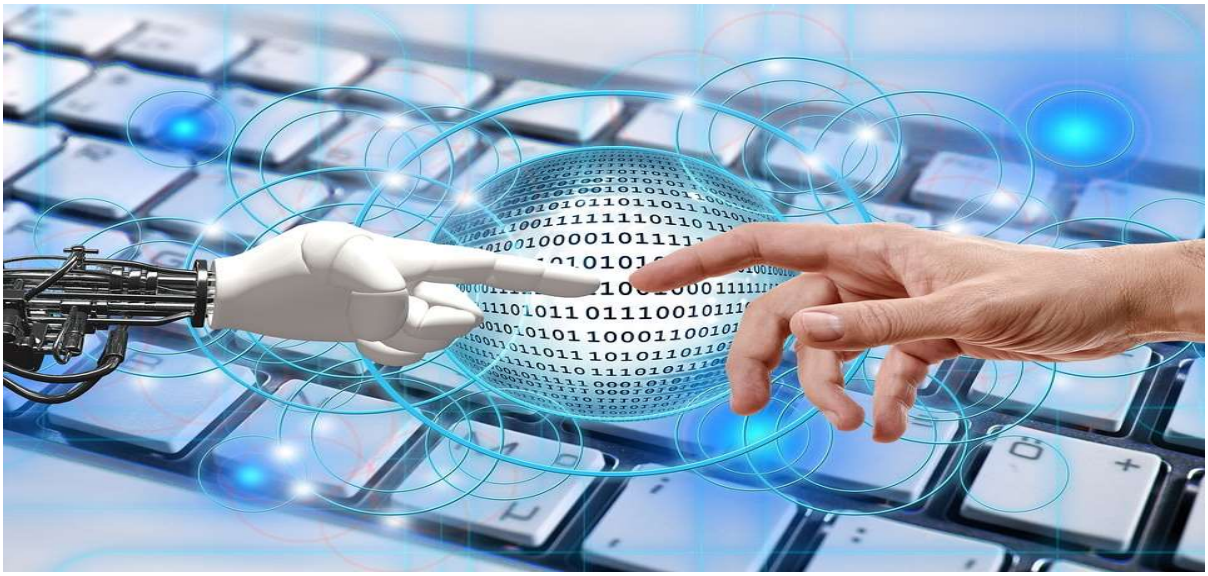
6. What class gives the text the primary bootstrap color?

7. What class is used for a container?

8. What class is used for a row?

9. What class is used for a column?

10. What class would you use to make a red button?



Introduction to jQuery

jQuery is one of the many libraries for JavaScript. It is designed to simplify scripting done on the client side. jQuery's most recognizable characteristic is its dollar sign ($) syntax. With it, you can easily manipulate elements, create animations and handle input events.

Scripting Tags and Document Ready

First in our HTML file, we need to include a script element on top of our page. In this script element we will include a document ready function. Without having the document ready function, our code may run before the HTML is rendered.

```
<script>
$(document).ready(function() {

});
</script>
```

All jQuery statements start with a $, which is known as a dollar sign operator or bling. Then jQuery will often select an HTML element with a selector and then do something to that element.

```
<script>
$(document).ready(function() {
   $("button").addClass("animated bounce");
});
</script>

<!-- This takes the button element and applies a bounce to it -->
```

So break down what we did is we selected the element $("button") and then added some CSS classes to the element. jQuery's addClass() function allows us to add classes to elements.

We can also target particular elements with the classes they have. So lets say that some of our elements have the class well. Let's add some jQuery to that.

```
<script>
$(document).ready(function() {

   $(".well").addClass("animated shake");
});
</script>
<!-- just like in CSS when adding a class we just put a period before the class name -->
```

Along with classes, we can also target elements by their id. Just like in css, we just include the # in front of the id name.

```
<script>
$(document).ready(function() {

     $("#idName").addClass("animated fadeout");
```

```
});
</script>

<!-- Like we would in css we use # to target the name of the id. -->
```

jQuery functions

Now let's look at removing a class with jQuery. In this case we will use the removeClass() function.

```
$("button").removeClass("btn-default");

// This just removes the class btn-default from all the button elements.
```

Another function that jQuery has is the css function. It allows us to change the CSS of an element. So maybe we want to change an <h1> element to blue.

```
$("h1").css("color","blue");
```

One important thing to note is that the property and value are in quotes and are separated with a comma instead of a colon.

jQuery also allows us to change the non-CSS-properties of HTML elements. All we have to do is use the prop() function. In this example we will modify the properties of a button so that it no longer works.

```
$("button").prop("disabled",true);
```

JQuery allows us to change the text between the start and end tags of an element. We can even change HTML markup. All we need to do is use the html() function, and if you wish to just edit text then you can use the text() function.

```
$("h3").html("<h2>Hello</h2>");
```

Let's now look at the remove() function. This function will remove an HTML element entirely.

```
$("h3").remove();

// This example will remove all elements that are h3
```

Let's now take a look at the appendTo function. This function allows us to move elements to another element. So if we wanted to move a button from a div to a form we would do something like the following:

```
$("button").appendTo("form");
```

Besides moving elements, we can also copy elements from one place to another. This involves the function clone(). But in order to properly use the function, we have to clone it and append it. So let's clone a button from one form to another. One important thing to note is that we can chain together jQuery functions and this is known as function chaining.

```
$("#form_button").clone().appendTo("#formTwo");
```

```
<div id="form_container">
  <form id="signup">
    <!-- HTML stuff -->
  </form>
</div>
```

Every HTML element has a parent element form which it inherits properties. In the example above the form element has a parent of . And by using the function parent we can access the parent of whatever element you've selected.

```
$("#signup").parent().css("background-color","blue");

// This jQuery function looks for the parent of the element signup and then applies the css function on it so that we can change the background of the div element.
```

Like we have a parent() function we also have the children() function. This function targets all the children of an element, and then you can apply another function onto them.

```
<div id="parent">
  <div id="child1">

  </div>
  <div id="child2">

  </div>
</div>
```

This example above gives us a parent div and within it we have two child elements. So let's apply some css to the child elements using jQuery.

```
$("#parent").children().css("color","blue");
```

Additional CSS Selectors

```
$(".ClassName:nth-child(n)")

// This selector allows you to select all the nth elements with the chosen class or element
```

```
type.
```

We also have a way to choose all even or odd elements.

```
$(".ClassName:even")
// or
$(".ClassName:odd")

// jQuery uses zero-indexed which means the first element has a position of 0. So odd selects the second element and even will select the first element.
```

Basic jQuery Review Questions

1. What is jQuery?

2. In your HTML file where would you put the jQuery document ready function?

3. What does ALL jQuery statements start with?

4. How do you add a class to a function using jQuery?

5. How do you remove a class from a function using jQuery?

6. What function would you use to add a button to a form?

7. What is the element usually called that inherits properties from another?

8. How would you remove the class btn-default from all the button elements?

9. What attributes do we use to target elements by?

10. What do we use to call them attributes in CSS?

Introduction to SASS

Sass, or "Syntactically Awesome StyleSheets", is a language extension of CSS. It adds features that aren't available using basic CSS syntax. Sass makes it easier for developers to simplify and maintain the style sheets for their projects.

Sass can extend the CSS language because it is a preprocessor. It takes code written using Sass syntax and converts it into basic CSS. This allows you to create variables, nest CSS rules into others, and import other Sass files, among other things. The result is more compact, easier to read code.

There are two syntaxes available for Sass. The first, known as SCSS (Sassy CSS) and used throughout these challenges, is an extension of the syntax of CSS. This means that every valid CSS stylesheet, is a valid SCSS file with the same meaning. Files using this syntax have the .scss extension.

The second and older syntax, known as the indented syntax (or sometimes just "Sass"), uses indentation rather than brackets to indicate nesting of selectors, and newlines rather than semicolons to separate properties. Files using this syntax have the .sass extension.

***This section introduces the basic features of Sass**.*

Storing Data with Sass Variables

One feature of Sass that you won't find in CSS is that it uses variables. They are declared and set to store data. In Sass variables start with a $ followed by the variable name.

```
$main-fonts: Arial, sans-serif;
$headings-color: green;

//To use the variables

h1 {
  font-family: $main-fonts;
  color: $headings-color;
}
```

One useful part of using Sass is that it allows nesting of CSS rules. Below we have an example of normal CSS and then an example of using nesting.

Normal CSS

```
nav {
  background-color: red;
}

nav ul {
  list-style: none;
}

nav ul li {
  display: inline-block;
}
```

Nesting Sass

```
nav {
  background-color: red;

  ul {
    list-style: none;

    li {
      display: inline-block;
    }
f  }
}
```

Mixins

In Sass a group of CSS declarations that can be reused throughout the style sheet is known as a mixin. An important note is that newer CSS features take time before they are fully adopted and ready to use in all browsers. As features are added to browsers and CSS rules using them may need vendor prefixes. A good example would be box-shadow.

```
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;
}
```

But in order to use box-shadow we have to keep retyping those lines of code for every element that uses it. This is where mixins come into play.

```
@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;
}
```

We start with @mixin followed by a custom name. In this example $x, $y, $blur, and $c are optional. Using this mixin we can now call the mixin whenever we wish to use the box-shadow rule.

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

Adding Logic to Your Styles

If we want to test for a specific case we can apply the @if directive. And just like in JavaScript we can use @else if and @else to test for more conditions.

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

Using loops within Sass

The @for directie adds styles in a loop, very much like a for loop in JavaScript. @for is used in two different ways. “Start through end” or “start to end”. The main difference is that “start to end” excludes the end number while “start through end” includes the end number.

```
@for $i from 1 through 12 {
  .col-#{$i} { width: 100%/12 * $i; }
}
```

When the Sass is converted to CSS we get something like the following.

```
.col-1 {
  width: 8.33333%;
}

.col-2 {
  width: 16.66667%;
}

...

.col-12 {
  width: 100%;
}
```

Mapping over items in a List

The next directive we will be looking at is the @each directive. This directive will loop over each item in a list or map. On each iteration, the variable gets assigned to the current value from the list or map.

```
@each $color in blue, red, green {
  .#{$color}-text {color: $color;}
}
```

The syntax for a map has slightly different syntax.

```
$colors: (color1: blue, color2: red, color3: green);

@each $key, $color in $colors {
  .#{$color}-text {color: $color;}
}
```

Applying a Style until a condition is Met

We can use the @while directive to create CSS rules until a condition is met.

```
$x: 1;
@while $x < 13 {
  .col-#{$x} { width: 100%/12 * $x;}
  $x: $x + 1;
}

// This example will create a series of classes to be used within the html.
```

Using Partials

Partials are separate files that hold segments of CSS code. They are imported and used in other Sass files. This is great when you want to group similar code. For Instance all of your mixins. When naming our partials we start with an underscore (_) which will tell Sass it is a small segment of CSS and not to convert it to a CSS file.

```
// In the main.scss file we would include the following line to import the partial containing our mixins. _mixins.scss is the file we are importing.

@import 'mixins'

//What we notice is that the underscore is not included.
```

Extending One set of Styles into Another

Sass has a feature known as extend. This feature allows us to borrow CSS rules from one element and build upon them in another.

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

Let's say we want to have a new panel called big-panel, and it needs the same base properties as panel. This is when we would use the extend feature.

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

Basic SASS Review Questions

1. What is Sass?

2. How many syntaxes are there for Sass?

3. What are the syntaxes for Sass?

4. What's the difference between normal CSS and nesting Sass?

5. In Sass a group of CSS declarations that can be reused throughout the style sheet is known as a _____.

6. Can you put JavaScript in Sass?

7. What method would you use to loop over a list or map and on each iteration, the variable gets assigned to the current value?

8. What would you use to apply a style until a condition is met?

9. _____ are separate files that hold segments of CSS code.

10. What feature allows us to borrow CSS rules from one element and build upon them in another?

Introduction to React

React, popularized by Facebook, is an open-source JavaScript library for building user interfaces. It is used to create components, handle state and props, and utilize event listeners and certain life cycle methods to update data as it changes.

React combines HTML with JavaScript functionality to create its own markup language, JSX. This section will introduce you to all of these concepts and how to implement them for use with your own projects.

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.

```
const JSX = <h1> Hello </h1>;
```

JSX can represent more complex HTML, and one important thing to know about nested JSX is that it must return a single element.

Valid JSX

```
<div>
  <h1> Hello </h1>
  <h2> Hello </h2>
  <h3> Hello </h3>
</div>
```

But JSX elements written as siblings with no parent wrapper element will not transpile.

Invalid JSX

```
<h1> Hello </h1>
<h2> Hello </h2>
<h3> Hello </h3>
```

Adding Comments

JSX has a certain way for us to add comments into our code. So in JSX we add code by doing the following {/* */}. We need to type this around any comments we want to add to our code.

```
const JSX = (
  <div>
    <h1> Hello JSX </h1>
    <p> Hello Human </p>
    {/* This is a comment */}
  </div>
);
```

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.

Defining an HTML Class in JSX

In JSX you can no longer use the word class to define html classes. Instead we need to use the word className. It is important to note that we must use camelCase while using JSX. So click events like onclick would be onClick.

```
const JSX = (
  <div className='divClass'>
    <h1>hello</h1>
  </div>
);
```

Differences between JSX and HTML

In HTML almost all tags have an opening and closing tag, and they're also self closing tags. But in JSX things are a little different. Any JSX element can be written with a self-closing tag, and every element must be closed. So for example, the line-break tag must always be written as

Creating Stateless Functional Components

There are two ways to create a React component. The first is to use a JavaScript function. What this means is that it can receive data and render it, but does not manage or track changes to that data. The JavaScript function needs to return either JSX or null, and React requires your function name to begin with a capital letter.

```
const Component = function() {
  return (
    <div className='customClass'>
      <h1> Hello </h1>
    </div>
  );
};
```

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.

Creating a React Component

We can also use the ES6 class syntax to create a React Component.

```
class Kitten extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <h1>Hi</h1>
    );
  }
}
```

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 call's 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.

Let's look at how we can compose multiple React Components together. If we are building an app, we might have three components such as a Navbar, Dashboard, and Footer. To compose these components together we would create a parent component which would render all three of them as children. To render a component as a child in a React component, you need to include the component name in a custom HTML tag in JSX.

```
return (
  <Parent>
    <Navbar />
    <Dashboard />
    <Footer />
  </Parent>
)
```

So when React encounters a custom HTML tag that references another component like in the example, it will render the markup for that component in the location of the tag.

There are many different ways to compose components with React, and Component composition is one of React's features. When dealing with React, it is important to think of your user interface in terms of components. So break down your UI into basic building blocks, and the pieces become the components.

Rendering Class Components to the DOM

React components are passed into ReactDOM.render() differently than JSX elements. For React Components, you need to use the same syntax as if you were rendering a nested component.

```
ReactDOM.render(<ComponentToRender />, targetNode)
```

We can use this syntax for both ES6 class components and functional components.

Passing Props to Stateless Functional Component

In React, you can pass props or properties to child components. So if we can an App component which renders a child component called Welcome. We can pass property by writing something like this:

```
<App>
  <Welcome user='John' />
</App>
```

You can use custom HTML attributes that React provides support for to pass the property user to the component Welcome.

```
const Welcome = (props) =>
<h1>Hello {props.user}!
</h1>
```

Let's now look at how to pass an array to a JSX element. In order to pass an array we need to wrap it in curly brackets.

```
<Component>
  <ChildComponent colors={['red','blue','yellow']} />
</Component>
```

Doing this allows the child component to have access to array colors. We can use array methods such as join() when accessing the property.

```
const ChildComponent = (props) =>
<p>{props.colors.join(', ')}
</p>
```

React also lets you set a default props, and you can assign default props to a component as a property on the component itself, and React assigns the default prop if necessary.

```
MyComponent.defaultProps = {
location: 'Memphis'}
```

React has type-checking features to verify that components receive props of the correct type. You can use propTypes on your component to require the data type you want. It is considered a best practice to set propTypes, especially when you know the prop ahead of time. You can define propTypes property the same way you define defaultProps.

```
MyComponent.propTypes = {
  handleClick:
  PropTypes.func.isRequired }
```

In this example, PropTypes.func checks to see if handleClick is a function, and adding isRequired tells React that is a required property for that component.

Important Note

As of React v15.5.0 Proptypes is imported independently from React.

```
import React, { PropTypes }
from 'react'
```

Accessing Props using this.props

If the child component that you're passing a prop to is an ES6 component, then you have different conventions to access the props. First remember that anytime you refer to a class component within itself, you use the this keyword. So to access props within a class component you need to use something like this {this.props.data}.

```
class Animal extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
        <h1> A List of Pet Names </h1>
        <Cat name="Pippin"/>
    )
```

```
}

class Cat extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <h2> {this.props.name} </h2>
    )
}
```

React Component Terminology

Stateless Functional Component - any function you write which accepts props and returns JSX

Stateless Component - A class that extends React.Component, but does not use internal state.

Stateful Component - any component that does maintain its own internal state. You may see stateful components referred to simply as components or React components.

It is common practice to minimize statefulness and to create stateless functional components wherever possible.

Creating a Stateful Component

An important topic in React is state. State consists of any data your application needs to know about that can change over time. So what this means is that you want your apps to respond to changes and show an updated UI when it is needed. React gives us a nice solution for this.

To create state in a component, you need to declare a state property on the component class in the constructor. The property must be set to a JavaScript object.

```
this.state = {
  // describe state here
}
```

You have access to state objects throughout the life of your component, which means you can update it, render it, and pass it as props to a child component.

You can display any part of a state in the UI that is rendered. If the component is stateful, which means it will always have access to the data in state in its render. You can access

the data with this.state. It is important to note that in order to access the state value in the return, you must enclose it in curly brackets.

State is a very powerful feature of components in React. It will allow you to track data in your app and render a UI in response to changes in the data. React uses something known as a virtual DOM to keep track of changes behind the scenes. The virtual DOM will re-render all the components using that data when the state changes. React will only update the actual DOM when necessary.

```
class Cat extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: “Pippin”
    }
  }
  render() {
    return (
      <h2> {this.state.name} </h2>
    )
}

// In this example we create a state and then use it within the component.
```

You also have another way to access state within a component. Within the render() method, before the return statement, you can write JavaScript directly. This means you can declare functions, access data from states or props, and perform computations on this data.

```
class Cat extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: “Pippin”
    }
  }
  render() {
    const greeting = “Hello “ + this.state.name
    return (
      <h2> {greeting} </h2>
    )
}

// In this example a variable called greeting was created saying hello to the data within the
```

```
state. Then that variable is called upon in the return statement.
```

Changing a Components State

React also has a way that allows you to change the state of a component. The method for updating a component's state is called setState. When you call the method, you must pass in an object with key-value pairs. The keys are the state properties while the values are the updated data.

```
this.setState({
  Name: 'Frodo'
});
```

React expects you to never modify state directly so you need to always use this.setState() when the state needs to change. It is important to note that React may batch multiple state updates to improve performance. This means that the method setState can be asynchronous.

```
class Cat extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Pippin"
    }
  }
  handleClick() {
    this.setState({
      name:'Frodo'
    });
  }
  render() {

    return (
      <button onClick={this.handleClick}> Name Change </button>
      <h2> {this.state.name} </h2>
    )
}

// This example shows an example of pushing a button to change the name listed above.
```

Lifecycle Methods

React components have special methods that provide opportunities to perform actions at certain points in the lifecycle component. They allow you to catch components at a certain point in time.

componentWillMount()	Is called before the render() method when the component is being mounted to the DOM. It will log a message to the console.
componentDidMount()	This method is called after a component is mounted to the DOM. Any calls to setState() will trigger a re rendering of the component. Also a great place to attach event listeners for specific functionality.
componentWillReceiveProps()	This is called whenever a component is receiving new props. This method receives the new props as an argument. Written as nextProps. You can perform actions before the component updates.
shouldComponentUpdate()	Take's new state or props as parameters. Can use this method to compare props. Method must return a boolean value that tells React it must update.
componentWillUpdate()	Called just before rendering
componentDidUpdate()	Called after a component re-renders. Rendering and mounting are considered different things. When a page first loads the component is mounted. Then we re-render from there.
componentWillUnmount()	It is good to use this method to do clean up on the components before they are unmounted and destroyed.

Inline Styles

So now how do you apply styles to the React Code? You can apply a class to your JSX element using the className attribute and apply styles to the class in your stylesheet. But another option is to apply inline styles.

```
<div style={{color: "yellow", fontSize: 16}}>Mellow Yellow</div>
```

When using inline styles we have to notice that certain CSS properties ue camel case. For example, font-size is fontSize. Hyphenated words are considered invalid syntax for JavaScript object Properties. All property value length units are assumed to be in px unless stated otherwise. For instance, if you wished to use em then we would use the following code:

```
{fontSize: “4em”}
```

One thing you can do is include all of your styles within an object and then pass that onto the inline code.

```
const styles = {
   color: “yellow”,
   fontSize: 16
}

<div style={styles}> Mellow Yellow </div>
```

Advanced JavaScript in React Render Method

You can write JavaScript directly in your render before the return statement without having to insert it within curly braces. But if you wish to use a variable within the JSX code you need to place it within the curly braces.

Another useful application is using if/else statements to control what html is rendered.

```
render() {
if(this.state.boolean === true){
   return (
      <h1> Hello </h1>
   )
} else {
   return (
      <p> Hello </p>
   )
}
}
```

But this can cause problems when you have too many if/else statements. So we can use the following types of conditional statements.

```
{condition && <p>markup</p>}
```

So let's look at an example within our code.

```
render() {
  return (
    {this.state.boolean && <h1>Hello</h1>
  )
}
```

In this example, the markup that states hello will only show if the boolean value is true.

We can also use Ternary Expressions for conditional rendering. This method is very popular among React Developers. This allows you to apply logic within your JSX code. Remember that Ternary works like the following:

```
condition ? expressionIfTrue : expressionIfFalse
```

An example of applying a Ternary in the return statement would be like the following:

```
return(
  {(this.state.age > 18) ? <h1>You are Adult </h1> : <h1> You are a child </h1>}
)
```

One of the most popular ways to make conditional decisions about what to render and when is using props. Which means using the value of a given prop to automatically make decisions about what to render.

```
class Results extends React.Component {
  constructor(props) {
    super(props);
  }
  render() { return(
    <h1> {this.props.game} </h1>
  )}
}

class PinBallGame extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      Score:1600
    }
  }
  render() {
    return (
      {(this.state.Score > 2000) ? <Results game="New High Score"/> : <Results
game="You Lose !"/>}
```

```
    )
  }
}
```

You can also render CSS conditionally based upon the state of the React Component. To do this, you need to check for a condition and if that is met you modify the style object that is assigned to the JSX elements.

```
render() {
  let styles = {
    color: 'blue'
  }
  if(this.state.boolean){
    styles = {
      color : 'red'
    {
  }
  return (
    <h1 style={styles}> Hello </h1>
  )
}
```

Array.map() to Dynamically Render Elements

In reality, you may need your components to render an unknown number of elements. And a programmer has no way to know the state of an application until runtime because so much depends on a user's interaction with the program being written. So as a programmer, you need to write your code correctly to handle the unknown state ahead of time.

```
render() {
  const items = this.state.array.map((item) = > <li><h1>{item}</h1></li>);
  return (
    <ul>
      {items}
    </ul>
  )
}

// returns a list of dynamically populated items from an array.
```

One issue with how we dynamically rendered a number elements in the last example is that we did not include a key attribute. When you create an array of elements, each one needs a key with a unique value. React uses the keys to keep track of items that are added, changed, or removed.

```
render() {
```

```
  const items = this.state.array.map((item) = > <li key={item}><h1>{item}</h1></li>);
  return (
    <ul>
      {items}
    </ul>
  )
}

// In this example we included a key attribute within li.
```

Array.filter()

Using array.filter() is another useful tool to use when working with React. This allows us to filter the contents of an array into a new array. So let's say we wanted to show all online users for an application we would do something like the following:

```
render() {
  let online = this.state.users.filter( (user) => user.online);
  let render = online.map( (user) => <li key={item}> {user} </li>)
  return(
    <ul>
      {render}
    </ul>
  )
}
```

Basic React Review Questions

1. What is React?

2. What is JSX?

3. What syntax do you have to use to write JavaScript in JSX?

4. Write an example of a comment in JSX.

5. How can we render JSX directly to the HTML DOM using React's rendering API?

6. When we define a HTML class in JSX, What do we use?

7. What is the parent Component?

8. What are the children Components?

9. How do you pass an array to a JSX element?

10. Create a React Component called myComponent, that renders an h1 that says (Hello).

Introduction to Redux

Redux is a predictable state container for JavaScript apps. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. While you can use Redux with any view library, it's introduced here before being combined with React.

Creating a Redux Store

Redux is a state management framework that can with a number of different web technologies. In Redex there is a single state object that is responsible for the entire state of your application. The entire state of your app is defined by a single state housed in the Redux store.

The Redux Store is an object which holds and manges application state. There is a method called createStore on the Redux object which is used to create the store. The method takes a reducer function as a parameter. All you need to know at the moment is that the reducer function takes state as a parameter and returns state.

```
const reducer = (state = 5) => {
  return state
}

const store = Redux.createStore(reducer)
```

Getting the state from the Redux Store

You can also acquire the current state of the store object by using the getState() method.

```
const reducer = (state = 5) => {
  return state
}

const store = Redux.createStore(reducer)

let currentState = store.getState();
```

Defining a Redux Action

Redux is a state management framework, and a result updating states is one of its core tasks. In redux, all state updates are triggered by dispatching actions. An action is 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 an action may carry some data like a username. While data is optional, all actions must carry a type property that specifies the type of action that occurred.

```
let action = {
  type: 'LOGIN'
}
```

After creating an object, you then want to send it to the Redux Store. In Redux, you define action creators which is simply a JavaScript function that returns an action.

```
let action = {
  type: 'LOGIN'
}

function actionCreator() {
  return action
}
```

You have seen how to create an action and now let's look at the dispatch method. This method allows us to dispatch actions to the Redux Store. In order to dispatch the action, you use store.dispatch() and pass the value returned from an action creator.

```
let creator = () => {
  return {
    type: 'LOGIN'
  }
};

store.dispatch(creator())
```

So you've created an action and dispatched the action. The next thing to look at is how the redux store will respond to the action. The reducer function is responsible for state modifications. A reducer takes the state and action as arguments and it always returns a new state. This is the only role of the reducer. Another key principle is that state is read only. So the reducer function must return a new copy of state and never modify state directly. Redux does not enforce state immutability; you are responsible for enforcing it in the code of your reducer functions.

```
const reducer = (state = defaultState, action) {
  if (action.type == 'LOGIN') {
```

```
    return {
      login:true
    }
  } else {
    return defaultState
  }
}
```

The Redux store can be informed on how to handle multiple action types. A good example would be managing user authentication. You can have a single state object called authenticated which will represent if users are logged in or not. A good way to handle this is to use switch statements.

```
const authReducer = (state = defaultState, action) => {
  switch(action.type){
    case 'LOGIN':
      return {authenticated:true}
    case 'LOGOUT':
      return {authenticated:false}
    default:
      return state
  }
}
```

Common practice when working with Redux is to assign action types as read-only constants. Then just reference those constants whenever they are used.

```
const LOGIN = 'LOGIN'
const LOGOUT = 'LOGOUT'

const authReducer = (state = defaultState, action) => {
  switch(action.type){
    case LOGIN:
      return {authenticated:true}
    case LOGOUT:
      return {authenticated:false}
    default:
      return state
  }
}

// Rest of Code that needs to be changed.
```

Subscribe

The redux store object has a method known as store.subscribe(). The method allows you to subscribe listener functions to the store. These are called whenever an action is dispatched against the store.

```
store.subscribe( () => {
  console.log('MEOW')
}
```

Combining Multiple Reducers

The first principle of Redux is that one state object is used to hold all application states. You can define multiple reducers to handle different pieces of the application state. With those reducers can be composed into one root reducer, which then is passed into the createStore method.

In order to do this, we need to make use of the combineReducers method. This method takes in objects as an argument.

```
const rootReducer = Redux.combineReducers({
  auth: authenticationReducer,
  notes: notesReducer
});
```

Asynchronous Actions

At some point in web development, you will have to handle Asynchronous Endpoints. In order to deal with these requests, you will need to use Redux's Thunk Middleware. To include the middleware, you pass it as an argument to Redux.applyMiddleware(). This is then provided as an optional parameter to the createStore() function.

To create an asynchronous action, you then need to return a function that takes dispatch as an argument. Within this function, you can dispatch actions and perform requests. It is common to dispatch an action before initiating a behavior. Once you receive the data, you dispatch another action which carries the data as a payload along with information that the action is completed.

Never Mutating the State

Now let's look at the several methods of enforcing the key principle of state immutability in Redux. What this means is that you never modify the state directly. Instead, you return a new copy of the state. Redux does not enforce state immutability in its store or reducers. This falls on the programmer. In practice, the state will probably consist of an array or object, which are mutable. So let's now look at ways to return a new array without changing the original array.

Array.concat // Adding components to an array	Merges two or more arrays, Does not change existing arrays, returns a new array.
Spread Operator (...) [...arr, 'f'] // Adding components to an array	Takes original array and concatenates new elements.
Array.filter // Removing items from an array	Creates a new array from an original array. New Array contains items that match specified criteria
array.slice() // Removing items from an array	Returns a portion of the original array. First parameter is the index where the copy should begin. Second is where copy should end.

array.map()
 	
// Replacing items in an array
// Transforming an entire array
	You can use the map function to check each item and replace items that match criterion.
	You can also use map() to transform data without changing the original.

Copy an Object with Object.assign

This is going to look at ways to enforce state immutability when the state is an object. A useful object is the Object.assign() function. This will take a target object and source objects and then will map properties from the source object to the target object. This behavior is used to make shallow copies of objects by passing an empty object as the first argument, followed by the object you want to copy.

```
const Cat = {
  name: 'Waffles',
  age: 2,
  status: 'Hyper'
}

let newCat = Object.assign( {}, Cat, {status:'sleepy'} )

// newCat contains the original object but with the status property updated to be sleepy.
```

Basic Redux Review Questions

1. What is Redux?

2. What is the single state object that is responsible for the entire state of your application?

3. What is the name of the method to create a store?

4. How do you acquire the current state of the store object?

5. Create an action of login. (write out the function below)

6. Now dispatch that state to the store below.

7. What function takes the state and action as arguments and it always returns a new state?

8. Write a reducer function below.

9. What method allows you to subscribe listener functions to the store?

10. Write an example of that method below.

Bonus Questions (Functional Programming)

11. What is functional programming?

12. What functional programming method checks each item in an array?

13. What functional programming method uses a condition to apply an action to each item in an array that meets that condition?

14. What functional programming method returns a portion of an array?

15. What functional programming method will copy an array?

Introduction to React and Redux

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 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.

Example of a React component with a state with two props

```
class MyComponent extends React.Component{
        constructor(){
        super();
        this.state={
           Input: ' ',
         Messages: [ ]
        }
   }
        render( ){
        return(
        <div >
        </div>
   }
}
```

Creating a Redux store

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.

First, define an action type 'ADD' and set it to const ADD.

Next, define an action creator addMessage() which creates the action to add a message. You'll need to pass a message to this action creator and include the message in the action.

Then create a reducer called messageReducer() that handles the state for the messages. The initial state should equal an empty array. This reducer should add a message to the array of messages held in state, or return the current state. Finally, create your Redux store and pass it the reducer.

Example

```
//Action
const ADD = "ADD";
const addMessage = message => {
  return {
    type: ADD,
    message
  };
};

//Reducer
const messageReducer = (previousState = [], action) => {
  switch (action.type) {
    case ADD:
      return [...previousState, action.message];
      break;
    default:
      return previousState;
  }
};

//Store
const store = Redux.createStore(messageReducer);
```

Use Provider to Connect React and Redux

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:

```
<Provider store={store}>
        <App/>
</Provider>
```

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().

```
const state = [ ];

const mapStateToProps = (state)=>{
  return {
    messages: state
  }
}
```

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.

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().
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:

```
{
 submitLoginUser: function(username) {
    dispatch(loginUser(username));
  }
```

}

Connect Redux to React

Now that you have 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.

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.

Basic React Redux Review Questions

1. What is React Redux?

2. Create a Redux action that will add a name to a contact list.

3. Now create a reducer to handle your actions. (Remember to give it a default state)

Use a switch...

4. Create a store. Assign the createStore function to a variable called store.

5. Now create a reducer that combines multiple reducers.

6. What method would you use to subscribe listener functions to the store?

7. Now dispatch a name to the store using your Redux action.

8. What is the wrapper component that allows you to access the Redux store and dispatch functions throughout your component tree?

9. What function would you use to map your current state to properties?

Bonus. Create a jsx component and put your wrapper component inside it and give your wrapper component an attribute of store and assign it to store. (Remember in order to put JavaScript inside jsx you got to use { }'s) Next use ReactDOM to render the jsx component and document.querySelector('#app')