# WELCOME TO WEEK 3

Welcome to Week 3. In this week, you'll first learn about a concept in JavaScript called object destructuring. Then you will learn about different types of React components and their uses. You'll get acquainted with the concepts of the Virtual DOM and Single Page Applications. You'll set up Header and Footer components, and you'll implement routing between different views in your project using the library React-Router. You will also add breadcrumbs to your project with the help of Reactstrap.

This week, as with every week, you will multiple Code Challenges, and a Quiz, and you'll end the week with a workshop with your instructor and classmates. You will also work on your Portfolio Project this week, focusing on the UI and application flow and creating a new wireframe.

Remember to stay focused, study every day, and don't forget the 20 minute rule.

Happy learning!

# EXERCISE: PRESENTATIONAL AND CONTAINER COMPONENTS

In React, there is no formal classification of presentational vs container components. This is just conceptual terminology that has risen to describe how certain components behave

SLIDE:

Container Components

• Fetch and manage state data

• Pass props to presentational components for rendering

Presentational Components

• Render the UI – markup, styles - based on props passed to them

• Do not maintain their own state - only light UI state if any

Container components are responsible for fetching and managing state data and passing the state data to child components to be rendered. Container components don't do any rendering themselves. That means that not all React components are required to render a part of the UI - container components are focused on handling management of data instead.

Presentational components, on the other hand, are strictly focused on the UI, with styles -- colors, positioning, things of that nature.

Thus, presentational components will typically not hold any state data and only use the data handed down to them as props. When they do hold any state data, it would typically be just a little bit of data about the current state of the UI, such as the window size or if a modal is open.

Once again, this is not a formal classification in React, just a way to describe the kind of work that a component is doing. You'll find that some people use other terminology that refers to the same basic concept - for example, you might hear about fat vs skinny components, where fat means it stores state data and skinny ones don't, or smart vs dumb components.

Note that this is not meant to be a strict, dogmatic rule about how to develop components in React - components do not all have to be strictly presentational or strictly container in purpose. However, it is often a useful way to organize components.

In this exercise, we are going to insert a container component that will sit below the App component. We'll call it Main component.

So in the Components folder, go ahead and create a new file named MainComponent.js. And we're going to grab parts of both the App component and the Directory component and move them into this component.

Let's start with App.js. Go ahead and copy everything in App.js and paste it over to MainComponent.js.

In MainComponent.js, remove the line to import App.css.

Now, App.js and MainComponent.js are in different folders, so we're going to have to tweak the filepaths for the imports.

Remove the word "components" from the path to DirectoryComponent, then for the CAMPSITES data in the shared folder, we have to now tell the app to go down one directory then look for the shared directory, and we can do that by adding an extra dot here, so dot dot slash shared slash campsites. Just like when you're in your bash terminal, you use cd .. to go down one directory, that means the same thing here.

Make sure that your imports look just like this.

Then down here, we'll change the class name to Main.

Then remove the className="App" from this div here, but leave the div.

Then down at the bottom, change the export default to Main.

At this point, let's go back to App.js. Here, we'll actually delete everything inside the return except for this one div, and inside that div, we'll render the <Main /> component.

Then at the top, we'll need to import the Main component.

And we can get rid of all these other imports that we're no longer using here - the reactstrap imports, the Directory import, the CAMPSITES import.

Then we can also delete the constructor, because we're now storing the state in the Main component.

OK, we are finished with App.js.

Next we'll go to DirectoryComponent.js, and we'll grab the import for the CampsiteInfo component from here, and move that over. So this one we're cutting, not copying, we won't leave it in DirectoryInfo.

Then from the Directory component, we'll grab this state property for selectedCampsite, and copy it over to MainComponent's state as well.

We will also take the onCampsiteSelect method from the Directory component and move it to the Main component.

We are turning the Directory component into a presentational component, it's not going to hold any more state data. So we can also delete the entire constructor from the Directory component.

And we'll also grab the rendering of the CampsiteInfo component from the Directory and move it to the Main component, just below where we render the Directory.

Now the last thing we need to do is change how we handle the selected campsite, when it's clicked.

And this is probably the most complex part of this exercise, so I'm going to try to explain it carefully.

The DirectoryComponent no longer has direct access to the onCampsiteSelect method. But React lets us pass event handlers and other functions as props! So we can take this arrow function and move it to the MainComponent here, and pass an onClick event handler to Directory as a prop.

<Directory campsites={this.state.campsites} onClick={campsiteId => this.onCampsiteSelect(campsiteId)}/>

But we don't need to use the entire campsite object. Instead, we'll change this so we're using just the campsite ID. In the onCampsiteSelect method, we'll also change the variable name from campsite to campsiteId so we can be clear about that change we're making.

onCampsiteSelect(campsiteId) {

this.setState({selectedCampsite: campsiteId});

}

Then back in DirectoryComponent, we'll receive that onClick as a prop, and pass to it the id of the campsite that was clicked on. So when someone clicks on a card now, what will happen is that the onCampsiteSelect method back in the Main component will be able to fire, using the campsite ID. And it will then store the ID of the campsite that was clicked on, in the Main component's state's selectedCampsite property.

<Card onClick={() => this.props.onClick(campsite.id)}>

That means we need to also change how we render the campsite with the CampsiteInfo component. The CampsiteInfo component is expecting us to pass it an object, but now this.state.selectedCampsite is storing an id, and not an object. This is where the array filter method can help us.

Since we know the id, we can grab the entire array of campsites here with this.state.campsites, then we can filter through it for the object with the ID that matches the ID we want. Remember, filter always returns an array, and each campsite ID is unique, so this filter method should give us back an array that only has one object in it. Again, the CampsiteInfo component is expecting us to send it an object and not an array, so we'll extract the object by using the index like this, [0] at the end, and that will send that one campsite object to the CampsiteINfo component.

<CampsiteInfo campsite={this.state.campsites.filter(campsite => campsite.id === this.state.selectedCampsite)[0]} />

One more thing, we'll need to go back to DirectoryComponent.js and grab the CampsiteInfo component import and move it over to MainComponent. Also, we are no longer using CardText and CardBody here either, so we can remove those now.

so now we can run this and see that our app is still working. Everything we've done in this exercise has been just reorganizing the structure of our app, it shouldn't have changed the appearance -- oh except it has. When we moved the CampsiteInfo component from here (DirectoryComponent) to here, it's no longer being rendered inside a container. [open campsite component] Go ahead and open CampsiteInfoComponent.js.

You can see we've got a row, and we've got columns, but no container. SO we'll just add a container div here, and that should set us straight.

Now that's working. If you're using Git, this would be a good time to do a commit.

# 

# FUNCTIONAL COMPONENTS

Last week, you learned that there are two ways to define components - as a class component or a functional component. We have been defining components using class components so far, but now we will look at functional components.

Functional components are written just like a function - as a function declaration, function expression, or arrow functions, though typically you will only see them written as function declarations or arrow functions. Like with all components, they need to have the first letter capitalized. And they need to accept props as an argument.

Like class components, they will always return at least one React element, but they do not use a render method around the return. And they will never have a constructor method.

Another difference is that when you refer to props inside a functional component, you do not use the *this* keyword.

function Example(props) {

return <div>{props.someData}</div>;

}

The class component equivalent of this example functional component would be written like this:

class Example extends Component {

render() {

return <div>{this.props.someData}</div>;

}

}

These are equivalent.

Functional components cannot hold state nor use something called lifecycle methods - which you will learn about later.

Also, let me mention that in a recent version, 16.8, React added something called Hooks that changed things up a bit - with Hooks, functional components are now able to hold state and act more like class components. But this is still quite a new addition and will not be covered in this course. Check the additional resources for information on React hooks.

During this course, we will continue with the established way of using functional components only for components that do not hold state or use lifecycle methods.

In the next exercise, we will convert some of the class components we have written into functional components.

# 

# JAVASCRIPT: OBJECT DESTRUCTURING

We will be using something in JavaScript called object destructuring this week, so let's get acquainted with it now.

The destructuring assignment syntax was introduced in ES6 and gives us a shortcut for accessing properties inside objects, or items inside arrays, and putting them inside new variables.

We'll first look at the basic destructuring assignment syntax, then we'll look at how it works with function parameters.

Go ahead and pull up your browser's developer console and let's walk through it.

First, let's consider an object with three properties:

const hotel = {

id: 0,

city: "Honolulu",

name: "Hotel Honolulu"

};

Now let's say we needed to pull out each of these properties into a new variable. This would be the old way to do it:

const id = hotel.id;

const city = hotel.city;

const name = hotel.name;

And now if you type in each of those variables into the console, you can see that they grabbed the values from the object properties. As you'd expect.

Let's go ahead and refresh the console to clear out the variables. Then hit your up key a few times to bring the hotel object declaration back up, and send that through again.

Now I'll show you the new way, with destructuring assignment syntax. Or, because we're working with an object here specifically, object destructuring.

It's going to look a little funny, because we're using curly braces on the left side of the assignment:

const {id, city, name} = hotel; then press enter

And now if you check what's in each of these variables, you'll see that this worked exactly the same as doing it the old way I showed you before.

But it's just a lot faster.

We can do the same thing when we pass an object into a function. Let me show you.

Let's say we have a function that looks like this:

function logHotelInfo1(hotel) {

const id = hotel.id;

const name = hotel.name;

const city = hotel.city;

console.log(`${id}: ${name} located in ${city}.`);

}

And we can call it like this: sending the hotel object in as the argument:

logHotelInfo1(hotel)

Using object destructuring syntax, we can rewrite this like this:

function logHotelInfo2({id, name, city}) {

console.log(`${id}: ${name} located in ${city}.`);

}

And still call it the same way,

logHotelInfo2(hotel)

So when you see curly braces used on the left side of the assignment operator, or you see curly braces being used like this in the parameter list of a function definition, then that's your clue that destructuring is taking place, and this is how you read it. It's saying, here's an object, look inside it for these property names, then assign their values to variables using those property names.

This is not the only way that the destructuring assignment syntax can be used. This is just an introduction, enough to prepare you for the destructuring you'll be seeing in upcoming exercises. There's other ways to use the object destructuring syntax, and there's a destructuring syntax for arrays as well. If you want to get deeper into destructuring, there are links in the Additional Resources you can explore.

# EXERCISE - FUNCTIONAL COMPONENTS

Previously, we turned the Directory and CampsiteInfo components into purely presentational components that do not contain or manage any state. They simply render parts of the view using props that are passed down to them from the Main component.

Such components are great candidates to be turned into functional components. Let's start with the DirectoryComponent.js file. Here, what we will do is turn this class component into two smaller functional components.

The first one will be a new component named RenderDirectoryItem. This component will be responsible for rendering each card with different campsite details.

Functional components always receive any data passed to them as properties of a single props object, and that props object is passed in as the only argument, like this:

function RenderDirectoryItem(props) {

}

In this case, we are going to destructure the props object, so we'll do that right here in the parameter list, like this: [ **replace props with {campsite, onClick}** ]

Then a functional component has no constructor and no render method, it only needs a return statement.

And inside this return, what we'll do is pull out all this Card stuff and move it there. And since we destructured, we can no longer use this.props.onClick, we can use just onClick here.

Next, we'll change the Directory component to a functional component, and we'll just pass in props for this one, we could destructure it but we don't need to.

And for functional components, we no longer use the *this* keyword when accessing props, so we can delete that from here.

Then we'll remove the render() method completely.

And here, where we had the Card before, we will call the RenderDirectoryItem component, passing it both the campsite and the onclick props.

Lastly, this Component import was only being used for the class component, so we can remove that now.

That's it. We can stop and check at this point and make sure that everything is still working as it should.

Again, we're not making any visible changes. We're still only restructuring and optimizing our app and its flow of data. So it should work exactly as it did before.

Next, we'll refactor CampsiteInfoComponent.

We know we'll be removing the class components, so we can just get rid of the Component import right off the bat here.

We're going to split up this class component into three functional components, one for each of the methods inside this class component - renderCampsite, renderComments, and the render. So instead of one big component handling everything, we'll have three components that are each handling different parts.

Let's work this from the top. First, we'll delete this line for the class component, then we'll make this renderCampsite method into a functional component by adding the function keyword and capitalizing the first letter. Then it will receive campsites as a property of props, so we need to destructure it from the props object using the object destructuring syntax.

And that's it, everything else can stay the same.

We'll do pretty much the same thing with the next one - we'll make the RenderComments method into its own functional component, and we'll destructure out the comments property from props.

Finally, we'll turn the render method into the CampsiteInfo component, and we'll pass in props. And again, we need to remove the *this* keyword here for functional components.

Then here, we are no longer calling methods of a class component, we'll rewrite that this here to use the new functional components that we created. We'll make sure to pass in the campsite and comments props respectively.

And that's it for this exercise. Make sure to test that everything is still working in your browser.

# REACT VIRTUAL DOM

Before we talk about the React Virtual DOM, let's refresh on the concept of the browser DOM - the Document Object Model.

I've mentioned before that the Elements tab in the developer tools window gives us a representation of the DOM in real-time.

But the DOM is not HTML. Nor is it JavaScript. It acts as an interface between the two. The DOM is an API, an application programming interface. It is a set of specifications on how programming languages and HTML can interact with each other. It maps HTML elements to a node-based, tree-like representation of HTML behind the scenes, an interface that programming languages can easily understand, traverse, and modify.

This is an example of some HTML, and its corresponding representation in the DOM. This representation is created by the browser using a technology called a browser engine.

Every time a webpage is loaded, your browser creates this abstract model and holds it in memory.

The DOM is the interface that makes it possible for you to refer to HTML elements and attributes using JavaScript object and property syntax.

For example, you might create a new image element node in JavaScript like this:

const newImage = document.createElement("img");

newImage.src = "../images/someimage.png";

Then you could insert this node into the dom using a method like appendChild:

document.body.appendChild(newImage);

And it would finally be rendered by the browser into this HTML element:

<body>

...

<img src="../images/someimage.png" />

</body>

To repeat, there is an interface between a programming language like JavaScript and the actual code that gets rendered in the browser with HTML and CSS, and the DOM is that interface. It is cross-platform and language-independent, so it can be used in all operating systems, and you can interact with it using other languages, not just JavaScript, though the vast majority of the time it will be JavaScript.

In general, whenever you cause any changes to a webpage's view with JavaScript, then the affected nodes and their children in the browser DOM are recreated. Along with that, the browser view is also re-rendered graphically, and that can be an expensive process.

Now we can talk about the Virtual DOM and why it exists. The Virtual Dom is a technique not unique to React, React is one implementation of it.

The React application maintains a lightweight representation of the browser DOM that it keeps in memory. It's much faster and easier to update the virtual DOM, because it doesn't require re-rendering the browser view every time you do it.

With React, any changes to the DOM are first made in the Virtual DOM. So if a user clicks on a button and it changes the color of the webpage's background, for example, instead of manipulating the real browser DOM directly, this change first gets made in React's Virtual DOM.

When changes are made to the Virtual DOM, React compares the changes during a process called reconciliation, which is done through its reconciliation engine named Fiber. In this process, it looks for which nodes have changed in the DOM tree.

Then it figures out the most optimized way to patch the browser DOM with the changes, and to minimize the amount of re-rendering while applying the changes, for example by sending multiple changes in a combined batch instead of one at a time.

This is the main reason why you need to use setState for any state changes in React instead of setting the state directly with an assignment operator [slide here].

Because setState sends your change through the virtual DOM, if you don't use it, then you're sidestepping one of the most useful features of React.

You also learned before about setting unique key attributes when rendering lists of React elements using map. When doing so, you are helping out the React reconciliation process by giving it stable markers to track and look for when changes have been made, so that it doesn't re-render unchanged items unnecessarily.

Hopefully, this has made it a little more clear for you what the Virtual DOM is and why React uses it, and why it's important to use setState and keys - it helps React do its job: making updates to your web app in an optimized way.

# EXERCISE - HEADER AND FOOTER

In this exercise, we will be adding two new components - Header and Footer. And we'll say hello agaiin to our old friends Font-Awesome and Bootstrap-Social. Let's start with a couple of installations. ~~Remember, you can use either YARN ADD or NPM INSTALL:~~

yarn add font-awesome@4.7.0

yarn add bootstrap-social@5.1.1

Then open index.js and update it to import both:

import 'font-awesome/css/font-awesome.css';

import 'bootstrap-social/bootstrap-social.css';

Now we will add the Header component  
**Slide: Header Component**

In your Components folder, add a new file named HeaderComponent.js.

We'll start by adding the imports for React, Component, Navbar, NavbarBrand, and Jumbotron. two imports to HeaderComponent.

Then we'll set up a class component named Header, which of course needs a render and a return.

Then go ahead and open your MainComponent. We will be moving the navbar from it to the HeaderComponent.

And inside the return, we'll move over the Navbar component and everything inside it from MainComponent to HeaderComponent.

Then in HeaderComponent, let's remove this color attribute from the Navbar, we'll add a custom color later via CSS.

----

And we'll add a sticky attribute and give it a value of {top}, to make it sticky-top. This is all taken from the Reactstrap documentation.

<Jumbotron fluid>

<div className="container">

<div className="row">

<div className="col">

<h1>NuCamp</h1>

<h2>a better way to camp</h2>

</div>

</div>

</div>

</Jumbotron>

Next, we'll add a Jumbotron above the Navbar. We'll give it a "fluid" attribute, and that will make it a jumbotron-fluid class when it's rendered out. Inside it, we'll add a Bootstrap grid with a container div, then a row div, then a column div.

Then we'll add a couple of heading elements, h1 and h2, with the site name and tagline.

And now there's a problem, which is that we have two elements that we're trying to return from a component, both a Navbar and a Jumbotron.

And React only allows us to return one, remember.

One way to resolve this would be to wrap them in a <div> element.

But you don't always want to be throwing in div elements just for this purpose, when that div isn't accomplishing anything else. It's just an extra, unnecessary node for the DOM.

Fortunately, React has a solution for situations like these: a special JSX element called React.Fragment.

Which you can write like this: <React.Fragment> </React.Fragment>.

There's a shorthand for this you will encounter, where you can write it like this: <> </>

However, this isn't fully supported everywhere yet, so it's safer to use the longhand version for now.

We're almost done here. We need to export this component, we'll do that here at the bottom as the default.

--------

Then we'll go back to MainComponent, at this point we can get rid of the reactstrap imports since we're not using the Navbar here anymore.

Then we should import the Header component.

Then where we used to have the Navbar, we can add the Header component.

[Slide: Add Custom Style]

We'll pause here with components for a moment and go to our App.css file to add some custom styles.

First, I've decided I don't actually want all h5s to have the Lobster font, it's overkill, I only want to use that font on the h5s that are children of elements with the card-body class. So I'll modify this selector:

.card-body>h5

The greater than sign here says that I want this rule to be applied only to h5 elements that are immediate children of elements with the card body class.

And I also want the Lobster font on all h1, h2, and h3 elements, so I'll add those:

h1, h2, h3, .card-body>h5

And I'll separate them all with a comma to say that I want this rule to be applied to each of these selectors.

Then a separate color for the h2 element

And I'll just copy over the rest of the styles here from the written instructions, since you should already be familiar with what all these do from the Bootstrap course.

And let's take a look in the browser, and make sure that it looks like this, before you continue to the Footer component.

**[Slide: Footer Component]**

For the Footer component, again we will create a new file in the Components folder, and name it FooterComponent.js. And Footer will be created as a functional component, so we can just import React here.

Then we'll set up the Footer component, passing in props, and inside its return, we'll first set up a footer JSX element, lower case f, remember JavaScript is case sensitive so this footer element is different from the Footer component. We'll give it a classname of site-footer.

This will start to look quite familiar from your Bootstrap course. Actually, it is so similar that I'm not going to type it all out or explain it all, let's just copy and paste it. You'll find the code in the written instructions.

There's only a couple parts I want to explain. One is this: JSX has a few differences from HTML, and one of those differences is in the way that it treats whitespace. So I added spaces here using the curly braces to indicate that a JavaScript expression is being used, then a pair of single quotes around a space. There's different ways to create whitespace, I just find this one easy.

I'll show you why this was necessary. Let's go ahead and export this, then back in MainComponent, we'll need to import the Footer component, then render it here.

Then in the browser, you see how there's spaces between the social icons? If I take out these curly brace pairs, you see that the space goes away. And it doesn't matter if I hit enter or type in spaces, JSX won't add a single space like HTML would when you do that. So that's why I inserted spaces manually here.

The other thing I want to point out is that here, I turned these i elements into self-closing ones, where in HTML you would have to write it like this: .. [demo]. Just because JSX allows you do this when there's nothing inside the tags, and it's a little shorter to type.

That's it for this exercise, and this would be a good time for you to make a commit if you're using Git.

# OVERVIEW - REACT ROUTER

A big difference in how you will approach your React app includes not having different HTML pages for each unique view on your site. Instead, you will have just one webpage, index.html, and from it, you will use the React library to create an application that is launched from that one page.

[SLIDE]

In React, you will not have multiple HTML pages

React is used to build a Single Page App (SPA)

This is one of the core features of a Single Page Application, or SPA. You will learn more about SPAs in upcoming lessons. A routing library such as React Router is crucial to an SPA.

React Router is a popular third-party library in the React ecosystem. It provides us a collection of router components, route matching components, and navigation components - and handles routing requests for different views within our app.

It's also helpful with other concerns such as generating unique URLs that can be bookmarked. And it also handles going back and forward in the browser history.

[**SLIDE**]

Collection of router, route matching, and navigational components

Functionality:

* Provides ways to navigate to different views
* Provides a unique, bookmarkable URL for each view
* Handles browser history, ability to go back and forward

[React-Router Components]

Some of the common components you will use with React Router include:

<BrowserRouter> - Parent router that wraps around other React-Router components. It will be at the highest level of all the React Router components. It uses the HTML5 History API that lets you move back and forth in the browsing history.

<Route> - Renders the UI for a matching path.

* Use the *exact* attribute if you want an exact match

<Redirect> - Redirects to a new URL

<Switch> Groups <Route> components together in a way similar to a JavaScript Switch statement

<Link> - Creates links to a path, will render as <a> links in browser. Use this instead of anchor links so that your navigation is routed through React Router.

<NavLink> - A special version of <Link> that adds styles to the currently active link

This is a broad overview of the components you will be using from React-Router. I won't go into the details, as I don't think it will be useful without getting hands on. That's what you'll be doing in the following exercise. You will install React Router and begin to use it in your course project.

<https://reactjs.org/community/routing.html>

# 

# EXERCISE - REACT ROUTER

First, inside our project folder, we will install the package react-router-dom, using yarn add ~~or npm install~~:

yarn add react-router-dom@4.2.2

Next, in App.js, let's import the BrowserRouter component from react-router-dom (above App.css)

Then we'll render the BrowserRouter in the App component, making it the highest level component being returned from App, which gives it access to the Main component and its children.

Next, inside the Components folder, we will create a new file named HomeComponent.js.

In it, we will import React

Then create a functional component named Home, and all it's going to do right now is return a container div with a header-4 that says Home. And we'll export it as the default.

Now we will go set up the main routing code in MainComponent.js

**SLIDE: MAIN COMPONENT ROUTE SWITCHING**

In MainComponent.js, let's first import the Home component.

We will also import three components from react-router-dom here: Switch, Route, and Redirect.

Our goal here is to set up the brains of our router here so that it knows where to send users when they click on different links within our app.

Also - Before this, when we went to our app, it was by default showing us the Directory view. We're going to change that and set up a default homepage view.

So one of the things we will need to do first is get rid of the code for clicking and selecting a campsite in the directory for more information, because we won't be doing that here anymore. We'll remove the selectedCampsite property from the state, and we'll get rid of this onCampsiteSelect event handler.

Next, inside the render method for the Main component, above the return statement, we're going to create a locally scoped component named HomePage, using an arrow function. This component is defined inside of the Main component, so it's only going to be accessible inside the Main component, that's what I mean by locally scoped.

Right now, it's just acting as a wrapper for the Home component. We'll add more to this component later. And I'll talk more about why we're setting this up with an arrow function later, instead of as a function declaration.

Next, go ahead and delete the CampsiteInfo component here completely.

And remove the onClick handler from the Directory component.

Then here finally, we're going to set up some router logic.

Right under header, first we'll set up a Switch component. We'll end it right after the Directory component.

Then we'll use the Route component to set up two paths: the first will route any traffic that tries to go to the path /home to the HomePage component we set up earlier.

The second will also have a path, as well as a Boolean attribute named exact, to match the exact path.

That path will be directory. Next to that, we will set up a render attribute, and in it, we'll set up an arrow function that returns the Directory component.

Lastly, we'll add more more component here, a Redirect component with the attribute "to" set to the /home route. This Redirect component acts as a catch-all, sort of like the default statement in a JavaScript switch statement.

The Route components are acting like the case keywords in a JavaScript switch statement. Any routing request that comes through will go through this Switch component until it hits a matching Route. If there are none, it will end up at this Redirect component, which will send them to whatever path is here, which in this case is home.

We are now done with the Main component. Next, we will update the Header component.

**[leave CampsiteInfo import in Main component alone, will need it again later]**

**[Slide: HEADER NAVLINKS]**

Here, our goal is to add navigational links to the Navbar. We will also make the Navbar responsive. We'll do this with a combination of components from reactstrap and react-router. First, we'll need to add some imports.

import { Nav, Navbar, NavbarBrand, NavbarToggler, Collapse, NavItem, Jumbotron } from 'reactstrap';

import { NavLink } from 'react-router-dom';

To help us, we will need to add a few more components from reactstrap:

- import { Nav, Navbar, NavbarBrand, NavbarToggler, Collapse, NavItem, Jumbotron } from 'reactstrap';

- import { NavLink } from 'react-router-dom';

We will add Nav, then NavbarToggler, Collapse, and NavItem from reactstrap.

Then we will also add NavLink from react router dom.

[Responsive Navbar]

We'll now make the navbar in the Header component responsive. We will be adding a couple of items to its state to help with the Navbar toggler. That means we need to set up a constructor, with the props parameter then the super(props) method. Then we'll set up this.state as an object, with the property isNavOpen initialized to false.

Then we need to set up a method that will handle when the Navbar toggler button is clicked. We'll call it toggleNav, and when it's triggered, it will change the state using setState to the opposite of its current state. So if it's false, it'll be set to true, and vice versa.

Then one more thing we'll need to do in the constructor is to bind the event handler, by writing this. plus the event handler method name, then using a JavaScript method called bind to bind it to the component. This ensures that when toggleNav is called, then the **this keyword** inside it is referring correctly to this component.

At the Navbar component, we'll need to add the attribute expand="md". This is from Bootstrap, and will make the Navbar collapse and show the toggler for viewports smaller than medium.

Next we'll add the NavbarToggler component here, just above the NavbarBrand. This component will create a toggler button. We'll pass it an onClick event handler as a prop, with the value of this.toggleNav.

This will make it so that clicking on the toggler button will trigger the toggleNav method.

To the NavbarBrand class, we'll add the className of mr-auto to help with alignment.

Then let's go ahead and replace this NuCamp text with a logo image.

We're almost to the end here. Now we can start adding the NavLinks.

[NavLinks]

We'll wrap the NavLinks inside a Collapse component, and set its isOpen attribute to this.state.isNavOpen - so will be either false or true, depending on the current state.

And we need to give it a navbar attribute as well, as per the Reactstrap documentation.

Then we'll add a Nav component, with another navbar attribute.

Inside it, we'll set up 4 NavItems.

Inside each NavItem will be a NavLink with a className of "nav-link" and a to= attribute - this first one will be to the route for '/home/'.

Inside this NavLink, we'll also set up a Font-Awesome icon for Home, then the text Home.

<NavLink className="nav-link" to='/home'><i className="fa fa-home fa-lg" /> Home</NavLink>

We can copy this NavItem and make three copies.

Then we'll change the second one to link to the Directory route,

The third one to link to the /aboutus route

And the fourth one to link to the /contactus route.

With the corresponding font-awesome icons and text.

The navbar is all set now. We're not done with this exercise - we still need to update the links in the

Footer and tidy up a couple of loose ends. However, this is a good time to stop and make sure that things are working in your browser as they should.

The webpage should load with the Homepage initially, the Directory should take you to the view of the Directory component, and the About Us and Contact Us links will redirect you to Home since we have not configured routes for them yet.

In the Directory, you may notice that your code breaks if you click on any of the campsites right now.

We'll fix that, but let's go to the Footer component first.

**[ SLIDE: FOOTER LINKS]**

In FooterComponent.js, we'll first import the Link component from react-router-dom.

Then in the list of links here, we'll replace the anchor tags with the Link component, and a to attribute with the appropriate paths.

That's all for the Footer. You should be able to check in the browser and see that the links are working as expected.

Finally, let's open DirectoryComponent.js and fix that error I pointed out earlier.

From the RenderDirectoryItem component here, we will remove references to onClick entirely, both from the parameter list up here, and the Card component here.

Then in the Directory component, also remove the onClick from where it was being passed to RenderDirectoryItem.

Now in your browser, clicking on the campsites will no longer bring up an error. It won't bring up the campsite info component either, but we'll be fixing that soon with more React Router additions.

# OVERVIEW: SINGLE PAGE APPLICATIONS

In your previous course, Bootstrap, you had a different HTML page for each of the views on your website. But you probably noticed that all the pages had some things in common - a header, a footer, a navbar, et cetera. But every time you navigated to a different page, all those parts were being requested from the server and re-rendered again. This can be an expensive process.

A Single Page Application, or SPA, is a model where you just have one page, usually index.html.

And from that page, all the front-end application code, the HTML, CSS, and JavaScript for a website is downloaded locally to your browser when you first load the website.

Some examples of SPAs are Gmail, Google Maps, and Facebook.

Then when you switch between views inside the web app, instead of requesting the server for all the code for that view, the application code makes the minimum necessary server requests and re-renders just the components that have changed.

So you might still make server requests for back-end resources such as data from databases, but the front-end application code is already in your browser. So it's almost like a native or desktop application that's downloaded to your computer while you have it open in your browser.

PROs and CONS

SPAs typically offer a much faster user experience due to fewer server requests. One main downside to an SPA is that search engines are designed to index websites that have a traditional multi-page structure, so they can still have trouble crawling SPAs. Though their ability to do so is improving all the time, that means that SEO, search engine optimization, can be affected by having an SPA. That means SPAs are typically best for websites that are behind a login, where search engines would not index anyway.

Some other pros and cons:

PROs:

* Streamlined development: Easy to divide back end and front end tasks during development
* Can easily reuse the back end code for mobile app development
* Easy to debug in browser

Because the back end and front end are developed separately, development can be streamlined, and it's easy to divide up the back end and front end tasks during development. And it's easy to reuse the back end code for mobile app development.

SPAs are easy to debug, as you can easily monitor what's going on with developer tools right in the browser.

CONS:

* JavaScript is required to be enabled in the browser
* The initial download can be slow.
* Memory leaks.

SPAs will not work if the user has JavaScript disabled in their browser. The initial download of all the application code can be slow. This can be somewhat mitigated for very large SPAs by having the server side handle some of the rendering.

And you must be careful of memory leaks in your application, as they can be more harmful than in a traditional multi-page website.

While not exactly a "con" since there are solutions for it, one major consideration with an SPA is how to handle navigation, since you are not using the traditional browser navigation of going to different HTML pages. That includes not only being able to click navigation links to be directed to a different view, but also being able to use the browser's Back (and forward) button to go back in your browsing history. That is what React Router provides for React.

In the following exercises, you will begin to develop your React you will continue to add more functionality to your React Single Page App, and you'll learn more about using React Router as well.

# EXERCISE - ADDING TO THE SINGLE PAGE APP

[Slide: Add Contact Component]

With VS Code open to your project folder, in the components folder, go ahead and create a new file named ContactComponent.js. I will just copy and paste the code from the written instructions here. This code is for a presentational, functional component. For now it's just showing basic contact information. It's very much like the contactus.html page you had in Bootstrap. It's using Bootstrap grid and button classes, and making use of font-awesome icons as well. So all this should look pretty familiar. And of course you have your React import, and you export the Contact component at the bottom.

Next we need to tell our app how to route to this component. It's not an HTML page, you can't just go to the page in your browser. In MainComponent, we'll import ContactComponent, then we'll add it to our list of routes, so this is telling our app to watch the browser address bar, and whenever the route in that address bar matches /contactus, then show this component.

You notice this is different from how we routed to DirectoryComponent. For DirectoryComponent, we used the attribute "render" then gave it a function, which contains the component in JSX with the angle brackets. But with ContactComponent, we used the attribute "component" then gave it the component name, but without the JSX angle brackets. If you need to pass state data as props to the component that you're routing to, a good rule of thumb is to use this render syntax. If you're just routing to the component without passing along any state data, then you'll use this component attribute.

This is a good time to start our app and test it. I actually already have mine started, so I'll just bring it in. Now I can click on the Contact Us link in the navbar and good, it takes us to that page. And if you look up top, at the browser's address bar, you can see that it generated this URL, so I could bookmark this page, or just type this in at the top, and it would always take me to this same view. [demo it]

Next, we will update the shared data in the shared folder. In the written instructions, there are four files for you to download: campsites.js, promotions.js, partners.js, and comments.js. Place these inside the shared directory, replacing the existing campsites.js you had there from before.

And we'll just take a quick look inside each of them, to see that they all contain a single constant, each holding an array of, you guessed it, objects. The campsites array is pretty much the same as the one we had before, but we've just taken the comments out of it and put the comments into its own module. Notice that the campsites, promotions, and partners arrays all have just one object each that has a featured property set to true. That will become important in a moment.

Next, it's time to add some content to the home page!

We're going to start by opening MainComponent and importing the COMMENTS, PROMOTIONS, and LEADERS modules that we just added. Remember, this double dot here means that the main component will back out of the components folder, then look for the shared folder. [show this in the VS code explorer]

Next, we're going to pull all that data into the Main component's state property. Don't forget to put a comma between each property..

Then here inside this HomePage component, where we render the Home component, we'll pass in three props, one for each item that we want to feature on the homepage - the featured campsite, the featured promotion, and the featured partner.

And we can do this by using the filter method on the this.state.campsites, promotions, and partners arrays, and we'll evaluate the featured property on each object in each array, and the ones that have the featured property as true will be returned in a new array. And this should only return one object from each array, and we'll use the array index 0 to pull that object out of the array. Then we pass it to the Home component as props.

So there's actually a reason that this HomePage component is written with an arrow function instead of a function declaration here. It's because of a feature of arrow functions that we haven't discussed yet, and that has to do with the nature of the *this* keyword inside arrow functions. Arrow functions inherit the *this* of their parent scope. If this was a function declaration, then we could not use *this* inside it to point to the state of the parent class. So that's why here, we need to use an arrow function, so that we can easily get the data from the the Main component's state.

Now we can look at Home Component.

In here, we'll use Reactstrap Card components to display the featured items.

import { Card, CardImg, CardText, CardBody, CardTitle } from 'reactstrap';

Next, we'll set up a new functional component, a purely presentational component named RenderCard.

We'll destructure an object named item from props, right in the parameter list.

Then in the return, we'll set up a <Card>, a CardImg that pulls from the image property of item, and the name property of the item for the alt text, then a CardBody, a CardTitle with the item name, then a CardText with the item description.

\*\*\*\*\*

Then in the Home component, we'll get rid of this placeholder Home text and replace it with a Bootstrap grid row, then three columns. They'll each get a col-md class, which means they'll each take up a third of the row for medium and up viewports, but for viewports smaller than medium, they'll each take up a full row and stack on top of each other. They'll each also get a m-1 class,s for a little margin spacing.

Then inside each column, we'll use the RenderCard component, and here's where we pass the item prop, which gets destructured in RenderCard.

Remember, we passed the featured campsite, promotion, and partner objects into Home from the Main component.

And we're able to retrieve those using props.campsite, props.promotion, props.partner. So the featured campsite object gets passed in here, then it gets passed to RenderCard, which creates a Card with all the properties from the featured campsite object, the image URL, the name, the description. And same for the featured promotion, and featured partner.

And if we take a look at our website now, that's what you should see. You can see from going back into the data files that yes, this is the featured campsite, this is the featured promotion, this is the featured partner. And look, if you changed this featured property to true, that automatically changes the website.

That's all we have for this exercise. Go ahead to the next lesson to get deeper into React Router.

# REACT ROUTER PARAMETERS

In our web app currently, clicking on the campsites in the Directory is no longer showing the information for that campsite.

In the next exercise, you'll learn to use parameters in React Router to access that information;.

First, you'll set it up so that When you click on a campsite, your React code will automatically grab the ID of the campsite and add it to the path, as you can see up here in the address bar.

So these URLs are not hard coded, they are generated automatically based on the campsite data.

Then, with React Router, the Route component will use this ID part of the URL, which here is 3 (or whatever number), Then it'll send the campsite object with that ID to the CampsiteInfo component and render the information for that campsite.

To let the Route component know that this part after Directory is going to be a dynamic route parameter, we will use this attribute: path='/directory/:campsiteId'

This colon here will cause the Route component to grab whatever string occurs after directory, then store that string inside a Route parameter called campsiteId.

That Route parameter is a property of the Route component's state. Specifically, it's stored inside a state object called "match", as a property named "params".

We can look inside the React Devtools Components tool, and if I select this Route

component, then I can see it has a state, with a match object, and a params property, which is also an object.

And in that is the campsiteId, with a value of "0" (or whatever). If I click on a different campsite, you can see how that changes.

Then, this stored campsiteId can be used to pull up the campsite object with that ID, and that object can then be passed into CampsiteInfo component.

Now that you have an idea of what you will be doing, go ahead to the next exercise to put it into practice.

# 

# EXERCISE: REACT ROUTER PARAMETERS

Go ahead and open your project in VS Code, and open the Directory component. First we'll import Link from react-router-dom.

[Set Up a Dynamic LInk]

In the RenderDirectoryItem component, what we're going to do is surround everything inside the Card component with a Link component.

Remember, we've used the Link component before in the FooterComponent [open it] and in there, we linked TO a string, and here it was just a simple word like home, directory, aboutus.

But this time, what we want to do is create a link that looks like this when you want to view more information for the campsite with the ID of 1: /directory/1 [annotations]

And like this for the campsite with ID2: /directory/2

and so on.

And we want to be able to pass that campsite ID in dynamically.

This part does not involve route parameters, at this point all we are doing is creating a dynamic link using vanilla JavaScript.

So, [back to DirectoryComponent]

We're going to make this a link to, then for the string, we're going to use a template literal so that we can insert JavaScript into the link, so remember we need backticks for that, then we can say /directory/ then we're going to insert the campsite ID property in here like this.`

So now, whatever campsite you select, it's going to link to the path directory slash then that campsite's ID number. So that part is just creating the link, we still need to set up the router with route parameters to actually show the right view when a user clicks on that link.

Go ahead and open MainComponent, and in the group of routes here inside the switch component, we'll add another route. And this route's path will be directory forward slash colon campsiteId.

This colon here is very important, it tells the router that what follows this forward slash is going to be a parameter, and then it takes whatever that is and puts it inside this property, campsiteId.

Then in the Route component itself stores an object named match in its state, which has as a property an object named params, and this campsiteId gets stored as a property of that params object.

We'll make this Route render a component named CampsiteWithId.

And this match object gets passed to the CampsiteWithId component as a prop automatically, we don't have to specify it.

So next we need to create this CampsiteWithId component. We'll do this right inside the Main component.

And I'm going to need to refer to the Main component's state inside it with the *this* keyword, so I'll set it up as an arrow function. And it receives props from the Route component, and I'll destructure the match object out from props right here.

Then inside the return, we'll render the CampsiteInfo component, and we need to pass in a couple things as props, one is the selected campsite object, and the other is an array of all the comments for that campsite.

So remember, we have the full list of campsites inside Main component's state, and we can access that with this.state.campsites, then we filter, and we go through and look for the campsite object that has the ID that matches what's stored in match.params.campsiteId. And because this value is stored as a string, we need to convert it to a number. I'll use the unary plus operator here - this is a use of the + operator you haven't seen before, so I'll explain it..

When you have a number that's being stored as a string, and you want to convert it to a number, this is one quick way to do it. So let's look in the developer console. If I have a number as a string data type, and I put a plus sign in front of it, then it converts to a number data type.

That's all I'm doing here, just converting a digit inside a string, to a number. And since filter returns an array, and I want the campsites object, I'll use the 0 index to get the object.

Then for comments, I'll also filter out whatever comments match the campsiteID. And for the comments, I actually do want the whole array, not just a single comment, so I don't need to use the 0 index here.

OK, at this point, our code should be working, so let's pull it up.

From the Directory, I should now be able to click on each of the campsites and get the detailed information, including the comments.

Ah, but the comments aren't there. Let's look at CampsiteInfo component. And down here, it looks like we're trying to get the comments from the campsite object still, and remember, we moved the comments out of the campsite object, and we're sending the comments array in as its own prop, so we can get rid of the word "campsite" here and just have it as props.comments, and now that should be working.

And I should be able to go back and forward with the browser buttons.

And each of these should have its own bookmarkable URL at the top, and if I were to just type in the campsite ID I want to see up here, it should take me to it.

That's looking good.

# EXERCISE: BREADCRUMBS

We will go through and add "breadcrumbs", to the "CampsiteInfo", "Contact", and "Directory" components. Let's start with the "Directory" component.

We'll need to import "Breadcrumb" and "Breadcrumb Item" from "Reactstrap".

Then "here", we'll add a new "row div", just below the "container" div, followed by a "column" div. Then a "breadcrumb" component, which will wrap around two breadcrumb "item" components.

The first breadcrumb "item" component, will wrap around a link to "home", and the "text" Home.

The second breadcrumb item component will have the attribute "active", and it will just wrap around the word "directory".

Then below this, I'll also add a "heading" for directory, and a horizontal "rule".

We will more or less repeat this process in the other two components. I'm going to copy over the "reactstrap" and "react router dom" imports to "contact" component. But I'll get rid of the 'card' components, since we don't use those here.

Then, let's copy over this 'breadcrumb' component and everything inside it.

Then, we just need to change the text inside the "active" breadcrumb item, to "contact us". And "that's" all for the contact component.

Next, we can copy over the breadcrumb, breadcrumb item, and link imports to campsite-info-component.

Then, we'll also copy over this entire 'row' from contact component to campsite-info-component.

Once it's copied, we'll change this 'home' link to 'directory', and change the 'text' inside it as well.

Then for the 'active' breadcrumb-item, 'this' part will be a little bit different.

We're going to add the text for this 'dynamically', through the use of 'props' dot 'campsite' dot 'name'.

Then we'll use the same thing for the heading.

And I'll just indent this block over. Then let's go ahead and look at this in the browser.

Make sure to test in your browser that all the breadcrumbs are showing up, and all the links are working.

There's just one thing I'm not liking here, which is that the campsite name is being shown twice.

So let's quickly go back into campsite-info-component and delete the card title here.

We are now done with breadcrumbs and you've made it through the final exercise for this week.

Go ahead to the workshop assignment instructions. You can watch the video and read the instructions, but don't forget not to start on it until the workshop.

# ASSIGNMENT WEEK 3

**TASK 1**

Download AboutComponent.js and put it in the components folder.

Update MainComponent to integrate the AboutComponent into the single page application. You should be able to navigate to it by clicking on both the navigation bar and the footer.

**TASK 2**

Add a new functional component named RenderPartner inside AboutComponent.js.

In its parameter list, deconstruct an object named partner.

From the About component itself, find where a media list is being returned, and replace its