# Welcome to Front-End Web Development with React / Welcome to Week Two

**REACT COURSE OVERVIEW**

Welcome to Week 2 of your React course. This week, you will begin your React course project, which will closely resemble your Bootstrap course project.

Along with JavaScript, React makes use of something called JSX. JSX is a syntax extension to JavaScript that resembles XML or HTML syntax, but actually breaks down to plain JavaScript. You will begin learning about JSX this week.

You will also learn about a core concept in React - React components. Like Bootstrap components, React components are bundles of code that handle different parts of the UI. But instead of being written in mostly CSS, as Bootstrap components are, React components are written in JavaScript and JSX, and they're much more flexible and powerful.

You won't be saying goodbye to Bootstrap, because starting this week, you will install and use Reactstrap, which is a third party library that integrates Bootstrap with React. You will also use just regular old Bootstrap in some cases as well.

Some of the other key points you will be learning about this week include:

* How to store data in a React component using the state object
* How to pass data from a parent component to a child component using the props object
* Rendering data in arrays as lists with unique keys
* Importing and exporting modules
* React Developer Tools
* and more!

In the coming weeks, you'll also learn about routing and Single Page Applications, the MVC and Flux design patterns, Redux, networking basics, the Fetch API, JavaScript promises, and much more.

This week and every week, you'll have multiple Code Challenges as well as occasional Challenge Questions to answer, a Quiz, and an end-of-week Workshop where you'll review the week then work on an assignment together with your instructor and classmates. You will also begin to transitioin your Bootstrap Portfolio Project into a React version.

As always, I will remind you to stay focused, study every day, and don't forget the 20 minute rule. Happy learning!

# Introduction to React and Front End JavaScript Frameworks and Libraries

# SLIDE:

JavaScript Ecosystem

JavaScript in the center

By now, you have learned that there is the core JavaScript programming language, often referred to as vanilla JavaScript, the center of what's often referred to as the JavaScript ecosystem. Then there are many JavaScript libraries and frameworks, some very small, some very large.

Some are large enough to have what's considered their own ecosystems - other libraries that are built around them. And these are all typically open source and available online via sources such as NPM and Github. They're constantly being updated, and there are new ones that are being created all the time to meet changing needs and trends, or because people have figured out a better way of doing things then before. Part of being a web developer includes staying aware about the JavaScript ecosystem as it grows and evolves.

Both frameworks and libraries consist of code that somebody already wrote, that you can reuse. So what differentiates a framework from a library? The main difference is a concept called **inversion of control**. With libraries, you, the developer, have control over how you use it - you call the code when you need it. With frameworks, the framework already has a good idea of how it's going to do things, and the framework has places where it calls your code when needed, not the other way around. This is often referred to as the Hollywood Principle: "Don't call us, we'll call you".

Think about building something in the real world, like a shed. You're probably not going to grow a tree, chop the tree down, then make your own wooden beams from the tree. You could, but it's more likely that you'll go to the hardware store and buy the lumber that's already made for you, along with your tools, paint, whatever else you might need. You're still building the shed by hand, but you don't have to build every little part from scratch. You buy the pieces and the tools, but you decide how to put it together. You're still in charge. A library is like the hardware store.

With a framework, it's more like starting out with a blueprint already created for you to follow, or even the basic frame of the shed already built. You can still customize it and fill in the details, but the basic structure, the skeleton, is already there and you just have to flesh it in. .

Some popular front-end JavaScript frameworks include Angular, Ember.js, and Backbone.js. When you choose to use one of these frameworks, you will typically use it to architect your entire app using the blueprint that they give you.

React is considered a library. Other popular examples of front-end JavaScript libraries include jQuery and Lodash. You can use a little of them, or you can use a lot, it's up to you.

**React History**

React was first developed by Jordan Walke at Facebook. It was first used in 2011 for Facebook's news feed. In 2013, it was released as open source. It is designed for speed, simplicity, and scalability. When I say "speed", I mean two things - the speed of the finished application, but also the speed of development. It's designed to help the developer to write code faster.

Let's briefly go over some defining features of React and its approach to web development.

React is Declarative. React uses a Component-based approach. React plays well with other technologies.

React uses a declarative approach vs an imperative approach toward communicating with the browser DOM. That means you declare to React what you want in the user interface, and React handles updating the UI in a consistent way for you, managing all the creating and inserting and deleting of nodes. React acts as a mediator between you and the DOM.

That's in contrast to the imperative approach, where you would manipulate the DOM directly with native JavaScript methods like document.createElement, addChild, removeChild, setAttribute, etc. While this would give you more fine-grained control, it can also lead to inconsistency and redundancy. That's not to say that the imperative programming approach is necessarily worse, but the approach that React takes is declarative.

React uses a component-based approach. Like you saw in Bootstrap, In a Component based approach, we encapsulate the UI into small units called Components.

React plays well with just about any other technologies.. React makes no assumptions about the entire tech stack that you're going to use. It concentrates only on the user interface side of the story, and that leaves up to you how to implement the rest, such as what to use on the back end.

In this course, we will be using several other technologies along with React, including the third-party libraries Reactstrap, React-Router, and React-Redux. We will also use a web API called Fetch to communicate with a back-end server. But these are the choices that were made for this course, and to be clear, these are not requirements from React, and if you go on to use React in other projects, you may use something else.

Now that you've begun to develop a general idea of what the React library is, in the next exercise, you will create a new React app.

# 

# Getting Started with React

**[Slide: Yarn Package Manager]**

Yarn is a package manager like NPM, but one that was created by Facebook. ~~When it was originally introduced in 2016, it was much faster than NPM. but NPM has since caught up, speedwise, so it's a matter of debate whether it's better to use Yarn or NPM at this point.~~

**It gives you access to the same packages as NPM, and it has a slightly different syntax. For example, you will use the command yarn add to install packages instead of npm install. While in general, you are free to use either NPM or Yarn in other projects, we will use Yarn in this course, to give you exposure to using both. We will continue to use Yarn in React Native, then if you are enrolled in the Full Stack bootcamp, you will use NPM again for the Back End course.**

~~If you like, you can choose to keep using NPM instead. It will not make a difference here. But whichever you choose, yarn or NPM, you need to stick with the same one throughout the project, or problems can arise.~~

To install Yarn, go to the website linked in the written instructions and follow the steps for your OS. For MacOS, that means you will use Homebrew to install yarn, with brew install yarn. For Windows, download this installer and run it.

**[Slide: create-react-app]**

Create-react-app is a utility provided by the React team that will scaffold out the recommended starter files for a React project for you. It also handles installing and configuring several background utilities, including two very common ones called Babel and Webpack which help with compiling code and managing modules.

---

NEW for update to using npx for create-react-app

We will not need to install create-react-app. Instead, we will use a tool called npx that is installed along with npm. NPX is a package runner for NPM. It lets you run packages from NPM without installing them explicitly. We only use create-react-app at the beginning of our project, and then it isn't needed for the rest of the project, so we don't need to install it explicitly.

If you have installed Yarn, then create-react-app will recognize that and will scaffold out the starter files configured to be used with Yarn. If you have not installed Yarn, then it will default to the configuration to be used with NPM. **So make sure that you have installed Yarn before you proceed to the next part.**

Navigate in your filesystem to your NucampFolder on your desktop, then inside it, to the 3-React folder.

Open your bash terminal inside this folder, then type this at the prompt:

npx create-react-app@3.3.0 nucampsite

I'm just going to time travel into the future ... and here we are.

At the end of the installation, it very helpfully tells us to use the cd nucampsite and yarn start commands.

Let's go ahead and use the cd nucampsite command. Remember, CD stands for Change Directory.

In here, let's type LS to **list** the contents of the directory, **[ANNOTATION: ls]** You can see a number of files have been generated, including a package.json file, and the node\_modules folder, a src folder, a public folder, a default README file, and a yarn.lock file if you're using yarn. If you're using npm instead of yarn, here you would see a package-lock.json file instead of a yarn.lock file. **If you see a package.lock json file here instead of yarn.lock, that indicates that Yarn was not installed globally on your system. In that case, you should delete this nucampsite folder, go back to the Yarn installation step, and start over from there. If you have problems installing Yarn, please reach out for help.**

**Assuming that you do see a yarn.lock file here, let's continue. What you see here from the ls command is actually not everything in this folder.**

~~And actually, that's not all.~~ Type LS -A to see hidden files **[ANNOTATION: ls -a]** and you can see that there's also a .gitignore file and a .git folder here. ~~And you might also notice that there is this text here saying (master). that should have been your first clue to tell you that create-react-app automatically initialized this folder as a git repository for you.~~

The presence of this .git folder tells you that this nucampsite folder has been initialized already as a local git repository.

Not only that, if you check **git log**, it's already made your first commit for you.

Let's take a quick look at the contents of your .gitignore file.

We'll use a bash command called **less**, not to be confused with the CSS preprocessor less. This is for reading text files from a bash shell - type **less .gitignore** and now you'll see the first page of the .gitignore file. And we can see that create-react-app already put node\_modules and several other entries into this file for us, which is very helpful. You can press your space bar to see more pages if you like, or type q at any time to go back to the command prompt.

The one thing it hasn't done, and can't do for you, is set up a remote repository and link it to this local repository. If you intend to keep a remote version of this repository, then you will need to set that up and link it using the **git remote add** command as was explained in the Bootstrap course - if you need a refresher, you can revisit the Using Git section of your Bootstrap course.

And if you want to get out of viewing the text file, type the letter **q**, lower or upper case, **[ANNOTATE: q or Q]** doesn't matter, you don't have to hit enter, just type the letter q on its own and you'll be returned to the command prompt.

We can do the same thing with package.json: **less package.json** to see what's in it. Less is not a text editor, all it's doing is showing you what's in the file.

<https://github.com/facebook/create-react-app#whats-included>

You can see in the dependencies section that the react library was installed, along with react-dom and react-scripts. And you can see in the scripts section that the start script uses a tool called react-scripts. And you don't have to know a lot about how react-scripts works, but it will handle a lot for you, including starting up a development server similar to lite-server which you used in the previous course.

So let's try it out. Type q to exit from less, then type yarn start~~, or npm start if you're using npm~~. Let's do that now.

That should bring up your React app inside your default browser, and it should look just like this.. **[Bring in webpage]**

If you want to stop the server, as before, Ctrl-C will stop it. **[stop it]**

Coming up, we'll look more closely at the starter files that were scaffolded out for you by create-react-app. You'll learn how they work together and start updating them yourself.

<https://github.com/nitishdayal/cra_closer_look>

<https://medium.com/the-self-taught-programmer/what-is-webpack-and-why-should-i-care-part-1-introduction-ca4da7d0d8dc>

# 

# OVERVIEW - REACT APP STRUCTURE

Go ahead and open VS Code from inside the **nucampsite** folder you created using create-react app. We are going on a tour.

From your bash terminal in your nucampsite folder, use yarn start ~~or npm start~~ to start your development server and see your project in the browser live.

The page that you're pulling up here in the browser is by default the index.html file that's in the **public** folder. Let's pull that up.

If I right click and View Source in the browser, then I can see that they match.

But now that you're looking at the source, you might notice something that's puzzling.

This content that you're seeing here in the browser, isn't anywhere in this HTML source code.

That's weird, right? I mean, you can see it says "Learn React" right here, but if I go to the source code and search for "Learn React", that's not in there.

And actually it says right here, "This HTML file is a template. If you open it directly in the browser, you will see an empty page."

Let's that that. If I go to my File Explorer in WIndows, it would be Finder for MacOS, and I just open this index.html file directly - that was no lie, it is just an empty page.

So where is this content coming from, and how did it get here?

We are going to play detectives here, and that's the mystery we're going to explore next.

First, let's swing by the Elements tab in the developer tools and I want to show you something real quick.

Often, there can be confusion about what the document object model is, the DOM, and sometimes people get the HTML code for a page confused with the DOM. We can get a real life demo right now about how the DOM is not the HTML, or at least, not just the HTML.

You just saw the HTML file right? That's the static code, but the DOM is not static, it is subject to being changed. The best way to see a real-time representation of the DOM by looking in the ELements tab of the Developer Tools. And you can see right away that the Elements that are in the document object model for this page, right now, is not the same as the HTML. Some of it's the same. You can see there's the code from the HTML file, but if I click here in the body, and expand to see inside this <div> element with the "root" i id, I can see a lot more stuff in there - now I can see the logo image, and here's that Learn React text. None of that was in the HTML file, but it is in the DOM, and therefore you can see it in the browser.

But where does the code for this come from, if not the index.html file? Well, one big clue is pretty clearly staring at us in the face. It says here to look at App.js. Let's pull up App.js.

Now in App.js, we can see there is a function named App, which is capitalized. And it has a return statement that looks like it contains HTML tags. Maybe you remember that I mentioned something called JSX before? This is JSX, and it looks a lot like HTML inside JavaScript, but it has some weird things like these curly braces here, className instead of class.

Still, mostly it looks just like HTML. And most importantly, it looks like the HTML that we are seeing inserted inside index.html.

But still the question remains -- How did it get from here, in App.js, to index.html?

The answer is in this page - index.js. Create-react-app configured index.js to work along with index.html as the main entry point, so when index.html gets pulled up, the code in index.js runs too.

ReactDOM.render(<App />, document.getElementById('root'));

In index.js, the part I want you to pay attention to is this line that starts with: ReactDOM.render. And in here, you see the word App, formatted like it's a self-closing HTML tag. Then a comma, then document.getElementById('root'), which is a way to specify a DOM node that has the ID of root.

And if you were paying close attention, you've seen that ID before. Back in our index.html page, there was that div with the id of root. And if we look at the Elements tab again, we can see that here's the div with the id root, and inside it, there's the div with the App class. And if you look inside that, there's the rest from App.js.

So this line here in index.js is the final piece of the puzzle. This line is the portal from React to the browser. This is where all your React code gets wrapped inside a single parent component, which here is App.

Then the ReactDOM library, which is not a third party library but a core part of React, takes all your React code and renders it into the browser as child nodes of this root element.

You should now have a basic idea of how the different files of your React starter app fit together.

# 

# INTRODUCTION TO JSX

What are HTML tags doing in my JavaScript? This was my first reaction when I saw JSX for the first time in a .js file. As it turns out, it's all an illusion. Let me explain.

In vanilla JavaScript, you may have used a built-in method called document.createElement() for creating new HTML elements that could then be inserted into the DOM using methods such as appendChild.

The React library has its own special version of createElement that can be called using React.createElement(), which basically also creates a new HTML element that is specially configured to work with the React library. To use it, you would write something like this:

React.createElement("div", null, "Hello world!")

And this would call a method that would create a div element that contains the string "Hello world"

JSX is a syntactic extension to JavaScript that was created by React developers to make it easier and faster to develop webpages with React. Using JSX, you can write what *looks* like HTML inside your JavaScript, so you can write this:

<div>Hello world!</div>

And it's actually the same thing. It looks like HTML, but's compiled right back to the React.createElement code.

So really it's just a handy shorthand you can use when writing React. In other words: it's syntactic sugar. You could also think of it as a preprocessor - like Sass with CSS, JSX lets you write your code using one syntax, but before it's sent to the browser, it gets compiled to regular JavaScript.

In many cases, you can write JSX just like HTML. You can even use many HTML attributes inside the start tag, such as id. However, there are several differences. Some HTML attributes have different names in JSX, such as className instead of class and htmlFor instead of for, because class and for are reserved words in JavaScript that mean something different from what they mean in HTML You can embed JavaScript expressions inside JSX tags by using curly braces, and you'll see examples of that as we go. Another difference is that when an element has no content between the start and end tags, you can make it into a self-closing tag in JSX even if that's not possible in HTML - for example, in HTML, the <i> tag is written like this:

<i className="fa fa-phone"></i>

In JSX< you can write it like this:

<i className="fa fa-phone" />

There are some other differences, and we'll touch on more as we go. But for the most part, JSX is very similar to HTML.

<https://reactjs.org/docs/dom-elements.html#all-supported-html-attributes> - Supported HTML attributes

<https://react-cn.github.io/react/docs/tags-and-attributes.html> Supported HTML tags

# EXERCISE - INITIAL CONFIGURATION

install bootstrap reactstrap popper

add bootstrap.min.css import to index.js

remove importing logo

Add import of Navbar and NavbarBrand from reactstrap

Replace content of <App>

We will now begin to re-create the course project that we created in the Bootstrap course using React. However, we're not going to just throw out everything you've learned about Bootstrap. We will bring Bootstrap components into this course by installing a library called Reactstrap, and we'll also use the normal Bootstrap library to get access to the Bootstrap grid classes and more.

We have to install three packages for this. Let's go ahead and do that first thing - open VS Code in your React nucampsite project folder, then use the integrated terminal to first install bootstrap. You'll use the command yarn add to install the packages. And you'll need to refer to the written instructions for the version numbers of each package to install. Don't automatically use the one you see in this video, go to the written instructions for the correct version.

Next we will install reactstrap. Again, refer to the written instructions for the correct version to use, which may or may not match what you see here on the screen.

[while installing] Reactstrap will supply us with Bootstrap components that have been re-implemented as React components using JSX, with the same kind of styles and functionality as their Bootstrap counterparts.

One more - we will install the react-popper library as well. Again, check the written instructions for the version to install. Popper is required for some of Bootstrap's functionality.

Then we need to go to our index.js file and import bootstrap.min.css in there. Note that you will not need to specify a *from* path for this import.

Also, with newer versions of create-react-app, you'll notice that the index.js file will have this set of React.StrictMode tags. That's fine and you can leave them there.

import 'bootstrap/dist/css/bootstrap.min.css';

Next, we'll open App.js and here, we're going to import a couple of components from Reactstrap - Navbar and NavbarBrand.

**And we'll also import { Component } from react, and get rid of this logo import since we won't be using it anymore.**

**Then down here, we're going to replace all this App component code with code that's in the written instructions for this exercise. So go ahead and copy and paste that in.**

We'll talk more about what React components are in the next lesson. For now, just learn this: When we're using, or rendering, what are called React components, the syntax will look like HTML or JSX tags, with the angle brackets, like this.

The telltale sign that it's a component and not just a JSX element is going to be the first letter. If it's capitalized, you know you are dealing with a React Component. If it's lower case, you're dealing with a JSX element.

**So here, we brought in the Navbar component, and it's got angle brackets and it's capitalized.**

**It has the dark theme, and the contextual color info from Bootstrap, which if you recall is a cyan color.**

And the correct syntax for using this, and any of the other components from Reactstrap, you will find on the reactstrap documentation, which is linked in the Additional Resources for this exercise. (flip over to it)

**And we have aJSX div element with the container class from Bootstrap,**

but as I mentioned before, we can't say class= in JSX, we have to say className=, because class is a reserved keyword in JavaScript and means something different than the HTML class attribute.

And we also have the NavbarBrand component from Reactstrap and it has an href attribute and it just links to a forward slash.

Go ahead and save if you don't have AutoSave turned on - though I really do recommend turning Auto Save on if you haven't, remember you can do that through the File menu here -

And start your React app now from your bash terminal using yarn start ~~or npm start, whichever one you decided to go with.~~

And we can see the Navbar and NavbarBrand there. Make sure that you can see this in your browser as well.

# 

# JavaScript Import and Export

The import and export keywords let JavaScript files share data with each other. This is not a part of React but a part of JavaScript.

That data can be in the form of objects, primitives, and functions - that includes classes, as classes are technically syntactic sugar for functions.

There are two general kinds of exports. One is called a named export. The other is called a default export. There can be many named exports in a file. There can only be one default export per file. At the bottom of this App.js file, you see where it says export default App. That means that this function, App, is the default export for this entire file.

You've undoubtedly heard the term JavaScript module before, but now I can give you a definition. A JavaScript module is a JavaScript file that contains at least one export. So this App.js file is a module, but index.js is not, no export. [switch to index.js]

Here in index.js, you can see that the App function is imported using the function name, plus the location and name of the file it's from. The ./ here tells index.js to look for the file App.js in the same folder as itself. And the extension, .js is left out.

~~You see he re that some of the other imports have curly braces around them. Because App was defined as the default export from App.js, there's no need for curly braces around it.~~ And actually, we don't even have to use the name App here in index.js, because there's only one default export from the App module, we could name it anything here. we could name it Foo and it would still grab whatever data was in the default export,.

and you could just change this to Foo here and it would still work. But it would probably be confusing, so we'll just keep it named App.

With named, non-default exports, you have to use the curly braces when you import them. Let me show you - and you'll get more practice with this later, but let's just do a quick demo - you can change this default export into a named export by adding curly braces around the name and removing the default keyword.

OK, now our compiler is complaining that we're trying to import a default export in index.js.

So now in index.js, we have to surround the App here with curly braces to let the compiler know, we're looking for a named export named App from App.js, and now it's happy. And this time you can't just name this Foo or whatever, you have to use the exact name that you used in App.js.

But we'll go ahead and change all that back... If you tried this out with me, make sure you change it back too. So we're going to remove the curly braces, make sure it says App here and here, not Foo, and back in App.js, we're going to make App the default export again, then check and make sure that it's working in your browser.

That should help you understand what's going on here with these other imports. All these imports that are not using the ./ , they are coming from modules that are inside the node\_modules folder. And I mentioned that create-react-app by default installs a module bundler named Webpack, Webpack is actually doing some work behind the scenes to make it easy for you to import modules, so you can just say react here and it will be able to go to the node\_modules folder and find the module named react.js there. So that's why for the files that are not inside node\_modules, you will use ways to make it clear that you're not asking Webpack to go look in the node\_modules folder, like with the ./

# 

# INTRODUCTION TO REACT COMPONENTS

What are React Components? To begin answering that question, let's review what you've seen already about React Components. You used some React components in the previous lesson, taken from the Reactstrap library - Navbar and NavbarBrand.

You also saw the App component being used inside index.js - right here, when it was used in ReactDOM... .. Remember: You can tell when a React component is being rendered by two things. 1. It's inside angle brackets, like a JSX or HTML tag. 2) The first letter is capitalized. That's very important, it must be capitalized!

When you're conceptualizing React components, it may be helpful to think of it as analogous to a function. Like a function, a React component is a repeatable block of code. Also like a function, you have to define a React component with one syntax, then you call it via another syntax.

For example, you can write the definition for a JavaScript function like this, using a function declaration:

function sayHi(name) {  
 console.log(`Hi ${name});  
}

And that function definition by itself doesn't do anything until you call it somewhere else in the code, with the syntax for a function call, like this:

sayHi("Bob");

So you see there's a function definition, and a function call, right?

With a React Component, there's also a way to define the component, then there's a way to call the component. With components, we'll say that we *render* the component instead of calling, invoking, or using it.

Because that's the difference between a function and a React component, conceptually. They're both reusable blocks of code, but a React component's purpose is to eventually render a part of the UI in the browser, so it's like a specific kind of function.

Let's talk about how to define React components. There's a couple different ways.

One way is called a functional component, and we'll discuss that more later.

The way that we are using here with App is called a class component. We create a class component by using the class keyword, then the component name, which must be capitalized, then extends Component.

You may recall that extends is the JavaScript keyword that creates child classes from parent classes, and Component is a class that is imported from the React library, so we are basically creating a child class here.

Then inside curly braces, a React component will always have a return statement. For a class component, that return statement must exist inside a special method named render, which takes no arguments.

Inside that return statement, the component MUST return a single React element, typically using JSX.

If there's more than one line in the return statement, it's a good idea to wrap them all in parentheses.

But wait, you say. This App component is returning a div and a Navbar and another div...How does that make sense?

Look at it like this: There is only one element being returned here, this outer div that wraps around everything. It has a bunch of children inside it, but at this first level, there's only one element, and that's what matters.

*This* is fine. But if you had another div outside of here, or any other element, it would not work. (demo it)

OK, so we've seen a little about how components are defined. Remember, I said that like a regular JavaScript function, there are ways to define a component, and there's another way to render the component when you want to use it.

And to render a component, you've seen before that you write it like an HTML tag, but capitalized.

Now these Navbar and NavbarBrand components you're seeing being rendered here have both a start and end tag. Other components will be rendered using a self-closing tag, and you've seen that with the App component. So the App component is defined here (highlight it) in App.js, then if we go back to index.js, we can see where the App component is being rendered, right here, and the App component is self-closing.

All of the custom components that you will be writing in this course are going to be rendered with a self closing tag, like this.

To recap **[slide time]**: You can use the class keyword to define what's called a class component, along with extends Component. The other type of component is called a functional component, and you'll learn about those coming up. for both functional and class components, the component name must be capitalized.

It must have a return inside it, and for class components, that return must be wrapped inside a special render() method. And that return must contain a single React element, no more, no less, though that element can have all kinds of other code inside it, including elements and even other components. To render a component, you'll write it like an HTML tag, typically a self-closing one.

That's the how, now let's talk about the why. What are components good for, why do we use them? As I mentioned before, they're reusable pieces of code like functions, but with a specific purpose - they're for rendering different parts of the UI. So the App component is an exception to that, it's the wrapper component that goes around everything. But all the other components you will be using, you will see that they're all creating different parts of the UI, so that they can be moved around and used over and over again in a modular way to render the view that the user ultimately sees on their screen.

So the concept is similar to Bootstrap components, but it's all written in JavaScript, and instead of just using existing components like in Bootstrap, you can make your own custom components, or bring in components from third party libraries, of which there are many. So many. Just google "react component libraries" and you'll see what I mean. (show it) That alone should give you an idea of the popularity and the power of React.

In the next exercise, you will more about class components, and you'll learn about an important concept called state.

# EXERCISE - REACT COMPONENTS PART 1

Create folder named assets in public folder

Download images.zip file, unzip it and place the images inside the assets folder.

Add new folder named components in src folder.

Create new file named DirectoryComponent.js, with campsites data inside its state

Used it inside App component

In this exercise, you will create your first new React class component, using JavaScript, JSX, and Bootstrap together.

There are some assets you will need to download for this exercise.

Navigate to the public folder in your project folder and create a new folder named assets.

For practice using the command line, you can follow along with me here - after opening the course project folder with VS Code, inside the bash terminal, I'll use cd public to go into the public folder, then mkdir assets to create the assets folder in there.

Then from the page for this exercise, there's a link to download a zip file, images.zip. Go ahead and extract it inside your assets folder, so what you should end up with is an images folder inside the assets folder.

~~Next, back in the bash terminal, I'm going to back out of both the assets and the public folder at the same time. by typing cd ../.. -- this lets me go down two directories at once, so you can see I'm back at the root level of the project folder, nucampsite.~~

Next, back in the bash terminal, I'm going to back out of the public folder using cd ..

Now I'll go into the src folder with cd src

Then mkdir components

And of course there's other ways I can make these folders, like through VS Code or through Windows File Explorer, but I just wanted to walk you through doing it through the command line to give you more practice with the command line if you need it.

All the custom components we make in this course, we're going to put inside this components folder, just to organize it. It's not required, but it's a good idea.

In VS code, you can see that folder is now there, and inside it let's make a new file. Go ahead and give it the name it DirectoryComponent.js.

The first thing we will do is import React. We'll bring in the default react import, then we'll also bring in a named import called Component.

And then we're going to create what's called a class component.

class Directory extends Component.

As you know, this means that we're making use of JavaScript class inheritance to create a child class from this Component parent class.

Inside here, this time we will make a constructor.

In the class component you saw before in App.js, there was no constructor method. [flip over to it]

**So a constructor is not required for a class component, except in certain cases - we'll discuss the details of that later**. (re recorded) ~~That is true for just normal JavaScript classes as well - constructors are only required in a class definition if you are storing properties in it.~~

Now, the only argument we're going to pass into this constructor method is a special argument named *props*, Props is an important keyword in React, and we'll discuss it more later. For now, just know that whenever you create a class component in React that has a constructor method, you will need to include this argument named props, which is short for properties.

Then inside it we're going to add this line, super(props), which if you recall, means we don't have to say this.props = props in this constructor because that's going to happen inside the base component in the parent class.

Every time you make a React class component with a constructor method, you will use this, super(props) as the very first line of the constructor method, that is required by React.

The last thing we're going to do inside this constructor is set up a property named **state** that gets defined right inside the constructor. The state property is a special property in React, and it always needs to hold an object. So first we'll make an outer wrapping here of an object. We'll come back and fill this in later.

Every React component must return a React element. In a class component, you need to wrap your return statement inside a special method called render(). So we will set that up here:

render() {

return (

);

}

VIDEO FROM HERE

**~~Right now, it's not returning anything from inside the render method. Let's run this and see what happens...~~**

**~~Ooh, yeah that's not going to work. Let's return the bare minimum, a single React element - and this is how we can represent an empty div in JSX -~~**

**~~\*\*remember, you can't do this in HTML because an HTML div needs both a start tag and an end tag.~~**

**~~In the Elements view, you can see that rendered out in HTML as a pair of div tags.~~**

**~~Let's change this and put some text inside - Hello World!~~**

**~~And you can see that updated in the Elements view as well.~~**

This is the skeleton for a React class component. The constructor is sometimes required, sometimes not.

~~The constructor will always have props as its parameter, and it~~ *~~must~~* ~~call super(props).~~

~~You must have this class keyword, you must have extends, then the Component from React, and you must have this render method, with no arguments, and it~~ *~~must~~* ~~have a return inside it.~~

Now that we have the basic structure for a class component, we'll fill lit out. Inside the object that's stored in this state property, we're going to store data for different campsites.

We'll make a property named campsites. Then for the value of this property, we want to store the data for multiple campsites, and each campsite has different information associated with it like a name, an image path, an elevation... So the best way to represent that is going to be an array of objects.

~~But the state property needs to hold an object, not an array, but that's OK. We'll just nest it inside the object, by making the array a property of the object.~~

You can find the code for this array in the written instructions, so go ahead and pause now and copy and paste it over.

Remember, I mentioned before in Week 1 that arrays of objects tend to be a very common way that data is represented. And that it's useful to give each one a unique id. So you can see that in practice here.

~~Now, this state property is being defined right here in the constructor. The values for it are not getting passed in from outside. So that means when this component is rendered, it's hard coded to have this value for its state property.~~

return (

<div className="container">

<div className="row">

</div>

</div>

);

}

The render method must have a return inside it, and it has to return a single React element. Let's use a JSX <div> to create that, and we'll give it the Bootstrap class of "container", using className instead of class.

Inside that, we'll add a row div.

And inside this row, we're going to use a JavaScript variable, and the way we use JavaScript inside of JSX is to use curly braces, like this.

We haven't actually created this directory variable yet. We'll do that next

**This variable is named directory with a little d - not big D.**

**remember, JavaScript is case sensitive so this directory variable and the Directory component, with a capital D, are two separate things.**

.

[type const directory = ]

What we're going to set up inside this directory variable is an array of elements.

\*\*\*\*\*As it turns out, if you give React an array of elements to render, it will just pull out all the elements from the array and render them, it just knows what to do.

So, how are we going to create this array? Well, we're going to create it using the campsites data that's in the local state. So we know there's an array up in there, and we're going to grab that array like this: with this.state.campsites. Then we're going to use our old friend map, so dot map.

Then as you know, map needs a callback function that tells it what to do to each of the array items. And for the parameter name for the current item, we'll use campsite, that seems reasonable.

Then the fat arrow, and inside this callback arrow function body, we're going to have a return.

Don't confuse this return with the return from the whole component. This is only a return from this arrow function.

In this return, we're going to first set up a <div> that wraps around everything. We'll give it a Bootstrap class of "col", using className since this is JSX.

Then inside, we'll add an img, using the campsite.image, we'll set up a heading with the campsite name, and we'll grab the campsite description too. All using curly braces since we're inside JSX.

<img src={campsite.image} alt={campsite.name} />

<h2>{campsite.name}</h2>

<p>{campsite.description}</p>

So what's going to happen is that this map will go through all the campsites from the local state, and it'll make a new array where each array item contains this same set of JSX elements, but with a different campsite for each item.   
Then that whole array is going to be rendered inside this Bootstrap row.

Let's connect this component to the rest of our app, then we can take a look at the result in the browser.

We will need to export the Directory component as the default. Then,

We'll g o to the App.js file and import the Directory component at the top. Then inside the App component, we'll render it under the Navbar component, like this: <Directory />

And now we'll yarn ~~or npm~~ start to run the app.

It looks good!

But there's one thing. If you look at the browser console, you'll see this red warning message that says, each child in a list should have a unique key prop. And there are some warning messages that are OK to ignore, but this one, we won't.

Because this works, as you can see, but we're not following best practices here. Whenever you are giving React an array of elements to render like this, React wants you to give each item a unique key as an attribute to the topmost element in each item. That will help React keep track of this list of items and render any changes to it efficiently.

That's where the unique id that we gave each campsite can come in handy. We'll just add that to the div here, because it's the topmost element in each item, like this: key={campsite.id} -

And now React is happy, that warning is gone.

Also, aren't you glad that you learned how to use map last week? You'll come to see that map is one of the most useful methods to know, so if you're still feeling shaky on it, practice it more.

At this point, you can do a Git commit if you are using git.

We're at the end of this journey, but I'll meet you in the next video.

<https://reactjs.org/docs/lists-and-keys.html>

# OVERVIEW - COMPONENT STATE AND PROPS

In the last exercise [show Directory.js] you saw how we can store data inside the local state variable, then use that data to help form the UI.

This data is called local state because other parts of the app cannot see it. The only way that other components can see data stored as state is if we pass it.

To pass state data from component A to component B, you include that data as an attribute when you render component. Let's look at an example. We'll just tack it on to the bottom of our DirectoryComponent for now, but if you're following along with me, which you should, make sure to delete it later:

class ExampleParentComponent extends Component {

constructor(props) {

super(props);

this.state = {

number: 333

}

}

render() {

return <ExampleChildComponent number={this.state.number} />;

}

}

class ExampleChildComponent extends Component {

render() {

return (

<div>{this.props.number}</div>

);

}

}

We'll create one class component here that holds local state. And since it holds local state, it needs a constructor.

Then we'll create another class component that will receive that data. And this doesn't need a constructor because it's not going to hold state.

Then in the render method of the first component, we will render the second component, like this.

This second component is the child of the first component because it's being rendered inside it.

Now how do we pass this state data to the child component? Like this: Right after the name of the child component, we will create a custom attribute name, I'll just use number again. The name of this variable is actually up to me, it doesn't have to be number. But usually it will just be the same as the property name you're passing.

Then we're going to give it the value that we want to pass, and in this case, that's this.state.number.

Then in the child component, we can access that data like this: by using this.props.number.

Note that you cannot change this.props inside the child component. Props is a read-only variable.

This is important to React's one-way data flow architecture, and we'll talk more about that later.

Just for the sake of this demo, I'll connect these components to the main application by rendering the ExampleParentComponent in the DirectoryComponent, and let's see what that looks like. And you see that the number rendered here.

Now let me demonstrate for you that the props variable isn't just for state, you can pass other data, for example, a string literal. And you can even pass multiple props, like this: "Hello World". [greeting="Hello World"]

return <ExampleDataReceiver number={this.state.number} greeting="Hello World!" />;

Then you can access both the props in the child component like this.

<div>{this.props.number} {this.props.greeting}</div>

And again you can see that was rendered here [webpage]

In the next exercise, you will get a chance to practice using state and props, and you'll learn about a concept called lifting state up, and how to correctly update the state using setState. You will also learn to use the Reactstrap Card component, and you'll have a chance to try out an event handler.

-----

After a component's state property's values have been set in the constructor, it is important to never try to update the state directly. React provides us with a method named setState, and we must use it anytime you want to make any updates to the state.

# 

# EXERCISE: REACT COMPONENTS - PART 2

*Changed App component to a class component, added import of { Component } from react*

*Moved campsites constant array to a shared/campsites.js file, added comments, exported*

*Imported the array into App.js, App component state*

*Passed as props to Directory component*

*Converted Media elements to Cards.*

**[Slide: Lifting State Up]**

When you're designing your React app, one thing you'll need to consider is where to store state data - that is, in which component?. For example, you will often want multiple components to have access to the same read-only data.

At times, that will mean you need to move the state higher up in the component hierarchy. That's a concept in React called "Lifting State Up".

That's the first thing we will do. We will remove the campsites data from the local state of the DirectoryComponent, and we'll lift it up to its parent component, App. Remember, the App component is set up to render the DirectoryComponent inside it, so that makes App the parent and Directory the child.

Go ahead and open DirectoryComponent.js and we'll just delete the campsites array inside state.

And we'll actually go one step further. In the written instructions for this exercise, you'll find a campsites.js file. Go ahead and create a new folder inside the src folder named 'shared', and place the campsites.js file inside that, as I have here.

Inside the campsites.js file is an array of campsite objects, just like what you deleted from the Directory component's state. And you'll notice there's an export keyword in front of it, which means that this array can be imported into another file.

And that's just what we'll do in App.js. We'll import the CAMPSITES array from this filepath.

Then we will use that array to set the local state in App.

So now we have the data from campsites.js inside of App's state.

Next, we're going to pass it down as props to the Directory component.

And we'll do that by setting up a custom attribute here, campsites=this.state.campsites

Now back in DirectoryComponent.js,

**const directory = this.props.campsites.map(campsite => {**

**return (**

**<div key={campsite.id} className="col-md-5 m-1">**

**<Card>**

**<CardImg width="100%" src={campsite.image} alt={campsite.name} />**

**<CardImgOverlay>**

**<CardTitle>{campsite.name}</CardTitle>**

**</CardImgOverlay>**

**</Card>**

**</div>**

**);**

**});**

First, there is no campsites property in this component's state anymore, it's being passed in as props. So we'll change the state here to props.

After this, we should be able to start our app and see that it works the same as before.

Next, we'll add a Reactstrap Card component.

**[Slide: Reactstrap Card]**

To do that, the first thing we need to do is import it at the top. And we're actually going to be using a few different components from ReactStrap to put our Card together:

import { Card, CardImg, CardImgOverlay, CardText, CardBody,

CardTitle } from 'reactstrap';

Then down here, where we map out each campsite into JSX, we're going to add these components. I'm not going to spend too much time explaining how each component works, you're familiar with at least the concept of the Card from Bootstrap, and you can find information about the syntax in the ReactStrap documentation.

We'll add a card with 100% width, a CardImg component to replace the image element, and a card title component inside a card img overlay, and we'll get rid of the campsite description. All of it is wrapped inside a div that still has a unique key, and we'll change this Bootstrap column class to md-5 and add a margin-1 class to it.

So let's take a look at that now:

That's working. We'll add some styles to it later to make it easier to read, but let's finish up here in the DirectoryComponent first.

The last thing we're going to do in DirectoryComponent is set up an event handler for when one of these cards is clicked.

**[Slide: Add onClick Event Handler and Change State]**

Here's the plan. What we want to have happen is that when we click on one of these cards, we'll get a new card at the bottom that has more details about that particular campsite.

This is going to take a few steps to set up. Let's start at the top.

In the constructor, we will set up a new property in the state. This property, selectedCampsite, will keep track of whatever campsite was last selected by the user. We'll give it an initial value of null, for when nothing is selected yet.

Below this, we'll set up a method called onCampsiteSelect that we want to have fire whenever a campsite is clicked on. The campsite object will get passed into this method, and inside it, we're going to use this method that React gives us called this.setState, to change the value of the selectedCampsite property of state.

THis is important, so take note. IN React, you never want to update the state directly. For example, you would not do this:

this.state.selectedCampsite = campsite;

In fact, just by writing this code, you've triggered a warning in the console:

The constructor is the only place where you can assign a value to state properties directly. Outside of the constructor, always, always use setState, so that React can make updates to the DOM properly, because React has some special things it's doing behind the scenes and you don't want to bypass that. This is a part of React being a declarative, versus an imperative library, as I've mentioned before.

Next, we have to trigger this method somehow, and what we want is for this method to be triggered whenever a user clicks on a campsite card.

So let's go back to where we mapped out those cards. In each Reactstrap Card component, we'll add a React onClick handler - notice that the C is capitalized here - then we'll pass it an arrow function, which contains a call to the onCampsiteSelect method, passing in the current campsite object that we got from the props data.

There are different ways to set up event handlers in React, and this is just one of them. We'll get into more detail about event handlers later.

Now we have it set up so that whenever someone clicks on a card, that campsite gets set as the selectedCampsite in the local state. Now we need to actually display that campsite's details to the view.

To do that, we'll make a new method. Follow along with me - this method will be named renderSelectedCampsite, and we'll pass in a campsite object.

Inside, we'll first check:

if (campsite) -- this will make sure that the campsite has an object in it, because this condition would return false if the campsite value is null or undefined.

Then, inside the if block, we'll return a <Card> that includes not just the campsite image and name, but the description as well.

Outside of this if block, we'll set up a return for if we didn't make it inside this block - that is, if the campsite value was null or undefined or otherwise falsy:

And we'll return a simple empty div.

Finally, we need to actually call this method we just created.

We'll go to the return for the entire DirectoryComponent. Remember, these other returns are for methods that are contained inside the DirectoryComponent, they're just passing data around inside the component. When we're finally ready to break out of this component and send some data back to the parent component, that's gonna happen in this final return for the entire directory component right here. You can tell which one it is because it's the first top level return inside the render method. All these other returns are inside other methods.

So in this return, below the div where the array of campsites gets rendered, we're going to add a new div, as a new Bootstrap grid row, we'll add a column, then we'll call the renderSelectedCampsite method and we'll pass it the campsite object that was stored in the selectedCampsite property of state.

Let's check it out in the browser. And check that, yes, when you click a card, the campsite appears at the bottom with more details.

The last thing we will do is import a couple of Google fonts and add some styling to this website.

**[ADD FONTS AND STYLES]**

To add Google fonts to our website, we can use yarn ~~or npm.~~

Go ahead and stop the server, and use ~~either~~ yarn add ~~or npm install~~ to add these font packages:

yarn add typeface-lobster

yarn add typeface-open-sans

Then in index.js we will import these, like so:

import "typeface-lobster";

import "typeface-open-sans";

And we'll move the App.js import to below these.

--- test the below before recording, stop recording after deleting everything in App.css--

Then open App.css, and we're going to just delete everything in here and replace it with the following code:

We'll set the body font to open Sans

We'll set the h5 font to Lobster

And in the website [show] we can see that the fonts have change. We can also see that the campsite names are hard to read, so back in App.css, we'll select for just the h5s that are children of the element with the card-img-overly class.

We'll make the font color white, then give the h5s a semi-transparent black background.

And you can see that's easier to read.

Congratulations on making it to the end of the online exercises for this week!

# 

# REACT DEVELOPER TOOLS

I want to quickly introduce you to a browser extension called React Developer Tools. There are versions for both Chrome and Firefox, and you can either google for "React Developer Tools" to find it, or look for the links in the Additional Resources section below this video.

After you have installed the version for your browser, yarn ~~or npm~~ start your app. Then in your developer tools, you will have two extra tabs at the end: Components and Profiler. The Profiler tab is for auditing the speed of a React application. This is beyond the scope of this course, and we will not be looking at this tab. Let's take a look at the Components tab though, and here you can easily see all the components that are used on this page, and you can click through and get information about where state is being held, props, and other useful information. Please install this extension and just poke around with it on your own. It will help you understand the structure of the React application,a s well as debug it when you need to.

# 

# 

# WEEK 2 WORKSHOP ASSIGNMENT

SCRIPT:

Welcome to your Week 2 workshop assignment, your first assignment using React.

You've learned a lot this week about React, and it's time to put what you've learned into practice.

SCREENCAP:

At the end of your final exercise, you ended up with a card being rendered here at the bottom of the DirectoryComponent to display the selected dish.

Your mission is to take this card and extract it into a new component. It will be called CampsiteInfo.

This will be accomplished in three tasks. I will go over each briefly, but you can find full details in the written instructions. Note: Not every single step will be given to you. There are some parts you will have to figure out how to do yourself.

**Task 1:**

**Script:**

For Task 1, you will add a CampsiteInfo component in its own file named CampsiteInfoComponent.js.

In DirectoryComponent, you will import and render the CampsiteInfo component.

When you're finished with this task, test it in the browser. clicking on a campsite to select it won't show anything, but if you look in your React Developer Tools Components tab, you should be able to see the CampsiteInfo component there.

**---Written--**

Add a CampsiteInfoComponent.js file in the components folder.

Create a class component named CampsiteInfo inside it, and export it as the default.

Inside its render method, check if an object with the name "campsite" (passed in via props) can be evaluated as truthy (i.e.g is not null, is not undefined).

If so, then return an empty div that has the Bootstrap *row* class as an attribute.

If not, return an empty <div> with no classes applied to it.

Inside DirectoryComponent, go to the main return from the Directory component's render method. There, remove the second set of row and column divs, the ones that surround the call to the method renderSelectedCampsite, and remove the method call as well. Replace it all with a render of the CampsiteInfo component. Pass the selectedCampsite object as props to the Campsite component, using the attribute name of "campsite".

Make sure to import the CampsiteInfo component into DirectoryComponent.js.

At this point clicking on one of the campsites will not do anything because you are just returning an empty div. If you want to make sure it is working, you can open your React Developer Tools Components tab and you should see the CampsiteInfo component at the bottom. It will have a campsite prop of "null" at first, but once you click on a campsite, the corresponding object will populate the campsite prop.

**----**

**TASK 2:**

**--script--**

For Task 2, you will write a new method inside the CampsiteInfo component called renderCampsite. You will move over the <card> components that you used in the renderSelectedCampsite method in the Directory component over to this method, wrapping it all inside a column div.

At the end of this task, you should be able to select a campsite and see it appear at the bottom again.

**--written--**

In the CampsiteInfo component, add a method named renderCampsite(), above the render() method.

Give it a variable named "campsite" as the only parameter inside its parameter list.

Write a return statement for this component, with parentheses following the return. Inside the parentheses, add a JSX div element with the classes col-md-5 and m-1.   
Inside this div, copy over the <Card> component from the renderSelectedCampsite(campsite) method in the Directory component, and all the other Card components inside it. Make sure to import the Card components from Reactstrap.

You can now delete the rest of the renderSelectedCampsite method from the Directory component.

Inside the CampsiteInfo component's render() method, inside the empty <div> with the row class you set up earlier, call the renderCampsite method (remember to use *this*) and pass the campsite to it.

TASK 3:

**--script--**

For Task 3, you will write JavaScript to extract the list of comments from the selected campsite object and display it in the same row.

You will create a new method named renderComments to do this, and you will need to use the array map method inside it. When finished, your React app should look like this when you select a campsite:

And it should look like this when you shrink the viewport to below medium.

Be sure to work together, and don't forget the 10 minute rule during workshops. Good luck everyone!

**-written--**

-

Use this code:

{new Intl.DateTimeFormat('en-US', { year: 'numeric', month: 'short', day: '2-digit'}).format(new Date(Date.parse(comment.date)))}

to format the date correctly.