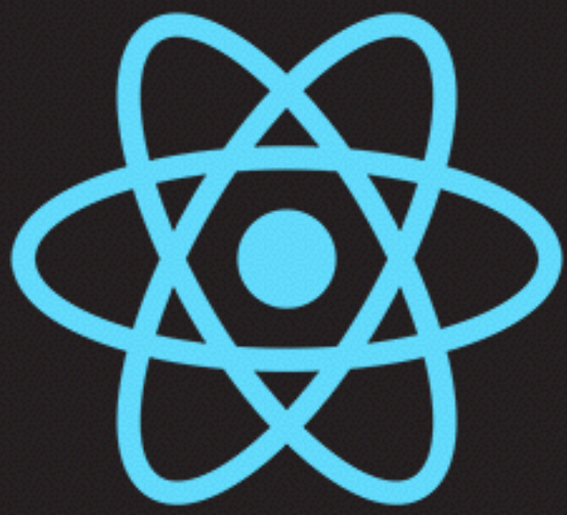
COLLEGIO TECHNOLOGIES INC.

REACT: INTRODUCTION TO COMPONENTS AND COMPONENT-BASED ARCHITECTURE



COMPONENTS AND COMPONENT-BASED ARCHITECTURE

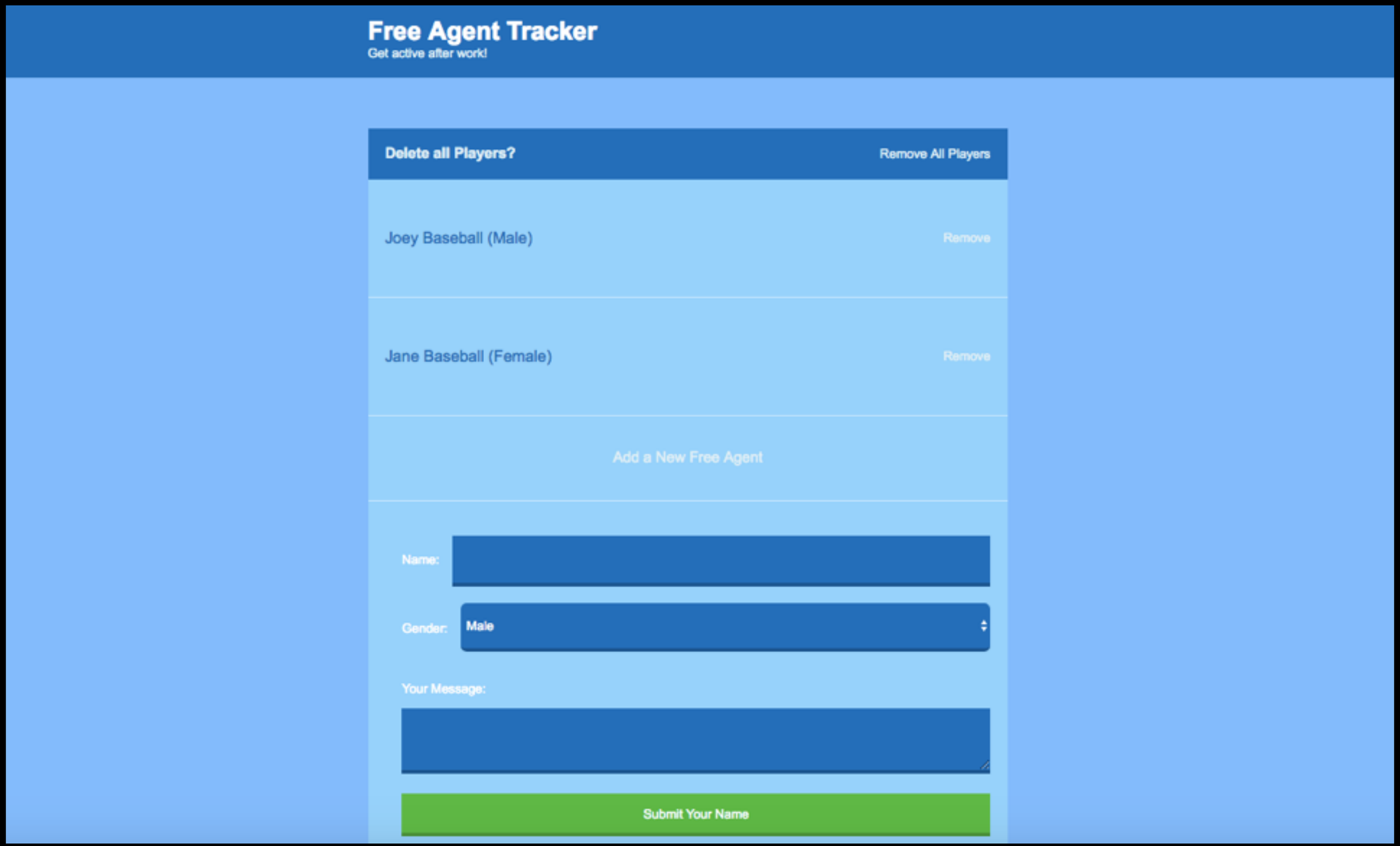
- One of the main features that makes React so powerful is it's wielding of a component-based architecture.
  - Components are ***small, logical, reusable*** chunks of code.
  - If implemented correctly, you should only write each unique element on your page once.

- Before writing any code inside of a React app, some thought should be given to the overall structure. It is much easier to identify the components of a wireframed app than it is to code on the fly.

- One thing to remember about React apps is that the entire app must be wrapped inside of a parent component.

- Let's review the free agent tracker, and break it down into it's respective components:

EXERCISE: IDENTIFY THE APP COMPONENTS



CREATING NEW COMPONENTS

- In it's basic state, a React component is an ES6 class instance that extends the ***React.Component*** class.
  - This class will allow us to perform a bunch of component-specific methods, such as rendering to the screen.

- For every component, you must define a ***render*** method. This is the method that displays the component.
  - The render function takes no arguments, and returns JSX.

CREATING NEW COMPONENTS

- When thinking about your overall component structure, you will first want to create a **root** component, which is that main component that all others will be directly or indirectly referenced from.
  - A good standard is to give this component the same name as the project or app, so in our case, we'll name this component **FreeAgentTracker**.
  - This is also the component that will be rendered using the **ReactDOM.render()** function.
- To ensure modularity, each component should be declared inside of it's own file, and included inside of any component files that require it. However, until we start working with Webpack in the next lesson, we'll just place them all inside of one file.
- Try it out! Let's create a couple of basic components and display them to screen.

EXAMPLE: HEADER AND BODY COMPONENTS

```
class Header extends React.Component {
    render() {
        return (
            <div>
                <h1>Free Agent Tracker</h1>
                <h2>Your after-work path to sports!</h2>
            </div>
        );
    }
}

class Body extends React.Component {
    render() {
        return (
            <div>
                <p>We will eventually place our code here!</p>
            </div>
        );
    }
}
```

EXAMPLE: FREEAGENTTRACKER COMPONENT AND RENDER STATEMENT

```
class FreeAgentTracker extends React.Component {
    render() {
        return (
            <div>
                <Header />
                <Body />
            </div>
        );
    }
}

ReactDOM.render(<FreeAgentTracker />, document.getElementById('app'));
```

COMPONENT PROPS

- So far, we've seen how we can render components to the screen, and also how we can use components inside of other components, but how do make them dynamic?
  - A component isn't very re-usable if it's static.
- This is where we use ***component props***.
  - Props are any data that is passed into the component from the source referencing it.
  - Props are passed into a component by including them inside of the component tag as a key-value pair, similar to an HTML attribute.

```
const theTitle = "Free Agent Tracker";
const theSubtitle = "Get active after work!";
class FreeAgentTracker extends React.Component {
    render() {
        return (
            <div>
                <Header title={theTitle} subtitle={theSubtitle} />
                <Body />
            </div>
        );
    }
}
```

COMPONENT PROPS

- Any props passed into a component are available via the ***this.props*** object.
  - For instance, if you pass a title attribute into the component, it will be available at **this.props.title**.
  - Although you can add static prop values to your component, most of the time you will be adding variable values, which are inject the same way as if they were displayed to screen. Unlike rendering, in this case you are allowed to pass in an object!

```
class Header extends React.Component {
    render() {
        return (
            <div>
                <h1>{this.props.title}</h1>
                <h2>{this.props.subtitle}</h2>
            </div>
        );
    }
}
```

DEFAULT PROP VALUES

- Adding props to our components allows us to make them dynamic, but what if a certain prop isn't provided, is it just undefined by default?

- React allows us to set default prop values for our components so that we can ensure that a value is always in a specific prop.
  - We can do this by defining the ***defaultProps*** attribute, which is an object that contains any default props that you want to include.
  - The defaultProps object allows you to have more control over your components by letting you set the default.

```
Header.defaultProps = {
    title: "This is the default title."
};
```

THIS.PROPS.CHILDREN

- In addition to standard props, you can set up components like a layout or visual component, with an opening and closing tag.

- Any information you place between the opening and closing tag will be available in the component via **this.props.children.**

- Let's set up our a component to use **this.props.children** to alert the supplied message to the screen:

EXAMPLE: THIS.PROPS.CHILDREN

```
class Body extends React.Component {
    constructor(props) {
        super(props);
        this.saySomething = this.saySomething.bind(this);
    }
    saySomething() {
        alert(this.props.children);
    }
    render() {
        return (
            <div>
                <button onClick={this.saySomething}>Say Something</button>
            </div>
        );
    }
}
class FreeAgentTracker extends React.Component {
    render() {
        const theTitle = "Free Agent Tracker";
        const theSubtitle = "Get active after work!";
        return (
            <div>
                <Header title={theTitle} subtitle={theSubtitle} />
                <Body>Hey there you!</Body>
            </div>
        );
    }
}
```

COMPONENT EVENTS

- When working with JSX, we just used global functions to set up action handlers for our events. However, now that we're working with React, we want to encapsulate everything within the component itself.
  - This is easy to do. Because a component is in essence a class, we can just add the appropriate event handler method to the component, and access it inside the JSX of the render() function.
  - Remember, because it's a class method, you'll need to use the **this** keyword.

```
class Body extends React.Component {
    saySomething() {
        alert("Hey There!");
    }
    render() {
        return (
            <div>
                <button onClick={this.saySomething}>Say Something</button>
            </div>
        );
    }
}
```

COMPONENTS: BINDING *THIS* TO METHODS

- In our previous example, we created a function called **saySomething()** that will alert the user when our button is clicked. This will work perfectly well inside of our React program; however, if we tried to access any variables located inside of the component's **this.props** object, we would receive an error.
    - This is because when we call the function inside of our JSX, we lose the scope of the component.

- In order to keep our component-specific variables, we need to bind **this** to our function call by using the **bind()** method and passing it **this**.
    - This will ensure that we can access the correct set of this variables inside of our function.

```
class Body extends React.Component {
    saySomething() {
        alert(this.props.welcomeMessage);
    }
    render() {
        return (
            <div>
                <button onClick={this.saySomething.bind(this)}>Say Something</button>
            </div>
        );
    }
}
```

COMPONENTS: BINDING ***THIS*** TO METHODS

- The previous method of binding this to your event methods works well, but also isn't the most efficient way to do this.
    - Binding can tend to be expensive with many re-renders, so it is best to take care of this a little as possible.
- We can also override the component's constructor function to add the correct binding.
    - The constructor function will be passed the **props** variable, and will re-initialize each method bound to **this**.
    - When declaring your functions this way, the binding on each function is only performed once, no matter how many times the view is re-rendered.

```
class Body extends React.Component {
    constructor(props) {
        super(props);
        this.saySomething = this.saySomething.bind(this);
    }
    saySomething() {
        alert(this.props.welcomeMessage);
    }
    render() {
        return (
            <div>
                <button onClick={this.saySomething}>Say Something</button>
            </div>
        );
    }
}
```

COMPONENT STATE

- In our previous example using JSX, whenever we needed to re-render our view, we just called a custom render method that re-ran ReactDOM.render().
  - This isn't an efficient way to run a react program, so let's look at how we can control updating the user-facing view using component state.
- The component state is a snapshot of current values inside of the specific component, that can be updated and re-rendered through different events.

COMPONENT STATE

- The requirements to using and working with system state are:
  - Set up the initial state object values. This is what is rendered initially by React inside of the component.
  - Change the state via an event. Once the state is changed, it will automatically re-render to the screen through React's default functionality.

- Initial state values are set up inside of the component's constructor function via a **this.state** object.
  - You can then access your state values through **this.state.yourstatevalue**

- You can update state values by using ***this.setState()***.
  - **this.setState()** should be passed an arrow function. The arrow function will accept the previous state as an argument, and return an object containing the specifically changed state values.
  - This will ***not*** override any state values that aren't returned by setState(), so you don't have to include any state values that haven't been updated.

EXAMPLE: COMPONENT STATE WITH THIS.SETSTATE()

```
// handleAddPlayer: Used to add a new player to the list of free agents
handleAddPlayer(player) {

    if (!player.name) {
        return 'Please enter a valid player name.';
    }

    // update the state
    this.setState((prevState) => {
        return {
            players: prevState.players.concat(player)
        };
    });
}
```

- Did you notice how we are using **.concat()** instead of **.push()** for the array to add the new item? This is because **.concat()** is an example of a ***pure function***.
    - A pure function is a function that returns an expected, repeatable value. Because .push() actually adds onto the original array, it is not repeatable because each subsequent call to .push() with the same value will produce a different result.

COMPONENT STATE: SOME POINTERS

- Remember, the main thing to remember about state (and props as well) is that it's a 'one-way street'.
  - Props are passed 'downwards' into sub-components.
  - A sub-component cannot directly affect the state of it's parent component.
- However, you need a way when you are changing state to be able to invoke a state change on a parent component.
  - This can be done by passing in our event functions as props, effectively being able to call them inside of sub-components.
  - By doing this, we can reverse the standard flow of component data by updating a parent component, causing a re-render of all child components where necessary.
  - Even though you cannot change the value of **this.props**, you can change the state of a parent component, which can effectively change this value on re-render.

EXAMPLE: PASSING EVENT HANDLER FUNCTIONS AS PROPS

```
// handleAddPlayer: Used to add a new player to the list of free agents
handleAddPlayer(player) {

    if (!player.name) {
        return 'Please enter a valid player name.';
    }

    // update the state
    this.setState((prevState) => {
        return {
            players: prevState.players.concat(player)
        };
    });
}
render() {
    return (
        <div>
            <Header>Free Agent Tracker</Header>
            <Body
                handleAddPlayer={this.handleAddPlayer}
                players={this.state.players}
            />
        </div>
    );
}
```

EXERCISE TIME: THE FREE AGENT TRACKER!

- Let's review what functionality we built into our free agent tracker:
  - The ability to add new free agents to the list via event methods passed down as handlers.
  - We can remove players from the list, either all at once or separately.
  - All functionality is encapsulated inside of it's component.

- Before we move on, let's review how we can simplify our code even further with Stateless Functional Components. Also, it would be nice to store this data so that if we refresh the page, the data won't wipe away., so let's do that with localStorage.

STATELESS FUNCTIONAL COMPONENTS

WHAT IS A STATELESS FUNCTIONAL COMPONENT?

- Well, the answer is in the name!
  - **Stateless** - this component doesn't carry an active state.
  - **Functional** - the component is built out as a function, as opposed to being a class-based function like we're used to.

- Even though we can't use state inside of these, we can still use props!

- Any basic component that has no need of modifying state and just renders some content to the screen can be used as a stateless functional component.

- It is much easier to use stateless functional components for purely presentational components.

EXAMPLE: STATELESS FUNCTIONAL COMPONENT

```
const Body = (props) => {
    return (
        <div>
            <PlayersList
                players={props.players}
                handleDeletePlayers={props.handleDeletePlayers}
                handleDeletePlayer={props.handleDeletePlayer}
            />

            <NewFreeAgentForm handleAddPlayer={props.handleAddPlayer} />
        </div>
    );
}
```

COMPONENT LIFECYCLE METHODS

- We've already seen an example of a component lifecycle method: the **render()** function.
- As the name implies, component lifecycle methods are invoked at specific points in the specific component's lifecycle.
- In addition to render, some important lifecycle methods are:
  - **componentDidMount()** - this method is invoked when the component is mounted.
  - **componentDidUpdate()** - this method is invoked whenever the prop or state values of the component have changed. You can view the previous props and state inside of this function.
  - **componentDidUnmount()** - this message is called when the component is unmounted from the view.
  - You can view all available lifecycle methods at **https://reactjs.org/docs/react-component.html#the-component-lifecycle**
- **Please Note:** These functions are not available with stateless functional components.

EXERCISE: COMPONENT LIFECYCLE

- Let's use the component lifecycle to store some of our data persistently using **localStorage**. If you are unfamiliar with localStorage, you can review it at https://www.w3schools.com/html/html5_webstorage.asp

- Using the component lifecycle's **componentDidUpdate()** and **componentDidMount()** functions, update and retrieve your free agent data so that you are not necessarily presented with an empty list on page load.

- **Tip:** In order to save arrays or objects in localStorage, it is necessary to use JSON for encoding and decoding using JSON.parse() and JSON.stringify().

RESOURCES

- **Component-Based Architecture:**
  - **Wikipedia:** https://en.wikipedia.org/wiki/Component-based_software_engineering
  - **Good article on understanding component-based architecture and it's benefits and drawbacks:** https://medium.com/@dan.shapiro1210/understanding-component-based-architecture-3ff48ec0c238
  - **TutorialsPoint - Component-based Architecture:** https://www.tutorialspoint.com/software_architecture_design/component_based_architecture.htm
  - **Video - Component-based Architecture Intro (not in React, but the principles still apply):** https://www.youtube.com/watch?v=X9hk56yyyu4&t=8s
- **React:**
  - **React.Component:** https://reactjs.org/docs/react-component.html
  - **Components and Props:** https://reactjs.org/docs/components-and-props.html
  - **React Events:** https://reactjs.org/docs/events.html
  - **State and Lifecycle:** https://reactjs.org/docs/state-and-lifecycle.html
  - **Stateless vs. Stateful Components in React:** https://code.tutsplus.com/tutorials/stateful-vs-stateless-functional-components-in-react--cms-29541
  - **Functional Stateless Component Tutorial:** https://javascriptplayground.com/functional-stateless-components-react/
  - **Video - React Component Patterns:** https://www.youtube.com/watch?v=YaZg8wg39QQ
  - **Video - Best Practices for Reusable Components:** https://www.youtube.com/watch?v=Yy7gFgETp0o

DISCLAIMER

- Copyright 2018 Collegio Technologies Inc.
- All content within this presentation has been compiled by Collegio Technologies Inc. All material has been sourced through open source resources and source code and as such conforms to Canadian copyright laws and regulations. All rights reserved.