Learn

Learn to explain class components, and use a class component to render some state data to the DOM

React comes with it a constructor class called `Component` that has some very special properties on it. This is a class like any other class that you've already become familiar with. React gives us a bunch of functionality through this single `base Component`. Using this is how we can scale out our static component based web pages to become fully functioning web-applications. To understand how to build out is fundamental and crucial for success when working with JS.

Overview

React gave us the idea of components as independent pieces of UI. And thus far, you have learned how to build out `functional components` for use in making multiple DOM elements. Now, we're going to be learning about the `React.Component` base class that allows us to use some of the methods that the React team has curated to tap into what we call the `Component Lifecycle`. These methods (known as life cycle hooks *more on these to come*) give us control over how our components work, and if we'd like to use them, we have to build out a class component that `extends` the `React.Component` parent class. Any time you see a line of code that looks like the following, you're using the React.Component parent class, and you have the ability to tap into these methods.

```
class FooComponent extends React.Component {}
```

By creating components as classes, you can set up a data object that your component is concerned with. This is done using `state` and setting up that object on our constructor method. Once we have some data that we can render out to the DOM, we need a vehicle that will allow us to render that data. This is achieved with the JSX method `render()` from within the life-cycle hook. We'll walk you through the steps below.

Declare your `class component` by extending the `React.Component` parent class. `class FooComponent extends React.Component {}`. Use the `constructor` function to set up some state. *because we're calling extends, we also need to call `super();` otherwise we won't have access the `this`* We need to render some sort of UI to the DOM. We do this by calling the life-cycle method `render`.

I like to remember these steps by referencing one of my favorite bands: Creedence Clearwater Revival (CCR), which stands for class, constructor, and render/return.

Declare your *class*, and extend the `React.Component` Base class.

```
class FooComponent extends React.Component {
```

1. Now we'll set up our *constructor* and add state.

```
constructor() {
  super();
  this.state = {};
}
```

1. *Render* some UI and *return* some `JSX`.

```
render() {
  return <div>Hello, I am Foo Component</div>;
}
```

Our final component should look like this.

```
class FooComponent extends React.Component {
  constructor() {
    super();
    this.state = {};
  }
  render() {
    return <div>Hello, I am Foo Component</div>;
  }
}
```

Now that we have constructed a skeleton for our Class component, it can be a bit more dynamic. The way we'll achieve this will be to use some data that we'll pre-define as some information we'd like our component to display. We'll then take that data and do this really cool thing called `interpolation` in order to present it to the DOM within some Text.

Components built out extending the Base `React.Component` class come with a bunch of benefits directly from the React API. A list of the benefits to what we get out of the Component class can be found here, in theReact documentation about class components. We will be discussing the `life-cycle` methods at another place in time, so don't worry too much about those for now.

For now, let's focus on a component caring about its own state (data) and managing that state in a reactive way. The `state` object that we set up on our `constructor` has a very React-specific way of doing things. It allows us to drive our UI using data. Again, think about Facebook here. You see a LOT

of data and interact with it all of the time when you're using the Facebook app. Because of the way we work with social media today, we expect this data the UI to represent that data in close to real-time. This is one reason why React is really good and how reactivity can be achieved.

Apply What You've Learned

Build out a class component that prints a message to the screen using a few DOM elements. We will hold a message on state, and print that message to the screen by selecting it an assigning it to a DOM element. Then we will take it a step further and pass that message down to another component using props.

Go ahead and navigate over to this Codesandbox, where we will write our React Code. CodeSandbox is an online editor that can be used to write React Code right away! I can't emphasize how cool this really is. For now, you'll just have to trust me.

You'll notice that we're getting an error on this page. As we begin to define our app class, elements will start to come to life on for us. For now, let's start by simply adding the class through CCR.

When you're done, your browser window should re-render without any errors. Your app class should look like this:

```
class App extends React.Component {
  constructor() {
    super();
    this.state = {};
  }
  render() {
    return <div>Hello From App!</div>;
  }
}
```

Now, let's add a property to our state data. Define a `message` property on the state object.

```
this.state = {
  message: "Hello from App State!!",
};
```

Now that we have the message on our component's state, we can use it through interpolation. In our render method, let's change the message inside of `div` to reference the state object. Remember the `this` keyword when pointing to an object on the Class constructor.

```
render() {
  return <div>{this.state.message}</div>;
}
```

Hooray! You've now built your first class component, and you're ready to rock n' roll.

Challenge

Take the functionality of this class component that we built earlier and extend it just a little bit. Declare a `Functional Component` called `RenderMessage` inside this CodeSandbox.

- Make sure you declare your Props Object that will be passed into this component.
- Return a `div` who's child is `props.message`
- Now inside of the `App` class pass in that `RenderMessage` component and pass down a message prop to `RenderMessage` . This message prop should be set equal to the message property on the state object.
- Once it's all wired up properly you've done it!

Dig Deeper

- Functional Components vs. Class Components
  Learn when to use them, so when the time comes, you'll know!

Learn to share data between components using state and props

We know about state already, and we know how to nest components. Now we're going to learn about how to share state with nested components using the react paradigm called props. Understanding how we link up data in a `uni-directional` (top to bottom) fashion will help you understand how react works generally.

Overview

Here is a link to the follow-along exercise from the video.

React is useless if we can't pass state around from one component to another. We use `state to props` to achieve functionality. Meaning that whatever is set on the state of one `Component` can be shared amongst all components by passing it down as props. Up until this point, we haven't had a significant need for this approach, but as your applications grow, you most certainly will. Now that we understand reactivity, we need to understand this `parent child` relationship in React in order to work with state more effectively and fluidly.

Because state is persistent as long as the component is on the screen, we can use it to hold on to memory for our application. Memory is any data that we pull in from a server, any edits from a submission form, or users interaction with your web page.

Remember that state is just an object that lives on the `class component's` constructor. We can take this state and pass it around as props. This will another reason why React is so powerful. What's great about this state object, is that when it changes, our component will re-render. We can also pass whatever data is found on this state object around as props to other components.

Remember that props are read-only, and components can use props to display information to the screen. One other principle that is built into React is that when a component's props are bound to the state of a parent component, the child component will re-render as well. This is again how we build out reactive data in react.

Now that we know more about state let's see how this works. We'll have a component set up to pass some Props around to a `functional component` .

```
class App extends React.Component {
  constructor() {
    super();
    this.state = {
      welcomeMessage: "world!",
    };
  }

  render() {
    return (
      <div>
        <h1>Hello, {this.state.welcomeMessage}!</h1>
      </div>
    );
  }
}
```

Now let's take this a step further and instead of rendering an `<h1>` lets actually put this message in another component and pass it through.

```
const WelcomeBanner = (props) => <h1>Hello, {props.message}!</h1>;
```

Now to pass this around properly, we need to re-factor our render function inside the `<App />` component.

```
...

render() {
  return (
    <div>
      <WelcomeBanner message={this.state.welcomeMessage} />
    </div>
  );
}

...
```

There we have it; now we are sharing data between a component's state and a component's props. This means that when the state object changes, so too will the props. We're going to combine that functionality in our follow along section.

Apply What You've Learned

Build out a few components and put some concepts together there.

- We're going to be updating some state on a parent component.
- That state will be wired up to a few other components as we pass the props around.
- We will also be passing around a few handler functions that help us update/delete our state.

Lets set up a form component that we can use to update our message component from above.

```
const WelcomeBanner = (props) => <h1>Hello, {props.message}!</h1>;

class App extends React.Component {
  constructor() {
    super();
    this.state = {
      welcomeMessage: "world!",
    };
  }

  render() {
    return (
      <div>
        <WelcomeBanner message={this.state.welcomeMessage} />
      </div>
    );
  }
}
```

Now let's build a form component that can handle some data defined on state, below on the child components.

```
const FormComponent = (props) => {
  return (
    <form>
      <input placeholder="change state" onChange={props.updateStateMessage} />
    </form>
  );
};
```

We're going to need to build out a change handler function on our `App` component that we can pass down to the form. We'll have to define the prop as `updateStateMessage` in order to make our `onChange` event handler work out properly.

```
...

messageChangeHandler = event => {
  this.setState({welcomeMessage: event.target.value});
};

render() {
  return (
    <div>
      <WelcomeBanner message={this.state.welcomeMessage} updateStateMessage={this.updateStateMes
    </div>
  );
}
...
```

Challenge

Using the following tools:

- Class component
- functional FormComponent, MessageComponent
- click, and change handlers
- `setState`

Build out a form that will allow a user to handle data. You'll need a button, input field, and some data-bound to a DOM element that displays what the user is submitting.

When a user clicks submit, show the data that's on state in an `alert` statement.

Stretch Loop over a list of items showing those items to the screen. (Can be a list of strings). When a user clicks submit, instead of logging the item, push an item into that list, and watch the magic happen.

Learn to respond to events triggered by user interaction and handle user input via forms in React

In Web development everything you do is based around the Document Object Model (DOM). The DOM has built into it an event loop that listens for changes across some part of the UI. To access that event loop in the Virtual DOM the react team put together what is called the Synthetic Event so that we can pretty much have access to all the events we would need to provide users with an interactive web application. Learn to harness the power of events in React and you'll be set to build out React Web Applications.

Overview1

Here is the link to the codesandbox follow-along exercise from the video.

Events in any programing language are unique to the language itself. When javascript runs in web browsers, it has a very eloquent way of handling events that are triggered by user interaction. This is one of the things that we, as web developers, should be most concerned with. How do we take in user input via forms and respond to clicks, mouse events, or scrolling? We will be learning how to respond to events within the React Ecosystem. Here is a statement straight from the React Docs.

> Handling events with React elements is very similar to handling events on DOM elements. There are some syntactic differences:
>
> - React events are named using camelCase, rather than lowercase.
> - With JSX you pass a function as the event handler, rather than a string.

Now, let's dive in and see what it is like to handle events in ReactJS.

It is important to understand that React has it's own "Synthetic Event" built into the API. This will mimic event handling as you're already used to when it comes to the DOM. When we bind up click/change/mouseOver handlers to DOM elements, we usually do so on the element itself, or we select the DOM element using vanilla JS to target by ID and append some function handler to that element.

```
<button onclick="getElementById('demo').innerHTML=Date()">
  What is the time?
</button>
```

We see stuff like the above code example all of the time. Now the purpose of today's exercises is to get us used to handling these types of events in the ReactJS ecosystem using Components/JSX and plain ol' JavaScript function handlers. This is where React starts to get really fun. React provides what we call the Synthetic Event, which is similar to the Event object in the DOM. This is how we interact with the DOM within the `React Virtual DOM` ecosystem.

Build a little button that we can click on to see what's going on under the hood.

```
class Button extends React.Component {
  render() {
    return <button onClick={// we'll fill this in}>Click Me</button>;
  }
}
```

So we have a simple button component that will return some Button JSX to the screen and mount up a button. Notice the `onClick` prop we used for our button. It's `camelCase`, which is different already from the standard `onClick` function that we normally would store as an attribute on a standard `button` element. This will be one of the major differences between React `Event Handling` and native `DOM event handling`.

Now that we have our button, we'll build a function handler that will allow us to interact with it. Let's assume that this function handler was found on a class component, so we can just add it as a method on that class.

```
clickHandler = (event) => {
  console.log(event); // this is the react Synthetic Event
};
```

The above code looks a lot like a typo, but inside of a class, this is valid JS. We'll chat more about this at length later, but now we need to bind this function up to our JSX element and use it to log out of the React synthetic event.

```
class Button extends React.Component {
  clickHandler = (event) => {
    console.log(event); // this is the react Synthetic Event
  };

  render() {
    return <button onClick={this.clickHandler}>Click Me</button>;
  }
}
```

Now we should see this on the screen, and when we click on it, we can see what the console is logging out for us.

```
▼ Class {dispatchConfig: null, _targetInst: null, nativeEvent: null, type: null, target: null…}
    dispatchConfig: null
    _targetInst: null
    nativeEvent: null
    type: null
    target: null
  ▶ currentTarget: function () {}
    eventPhase: null
    bubbles: null
    cancelable: null
  ▶ timeStamp: function timeStamp() {}
    defaultPrevented: null
    isTrusted: null
    view: null
    detail: null
    screenX: null
    screenY: null
    clientX: null
    clientY: null
    pageX: null
    pageY: null
    ctrlKey: null
    shiftKey: null
    altKey: null
    metaKey: null
  ▶ getModifierState: function getEventModifierState() {}
    button: null
    buttons: null
```

You should also see a bunch of warnings in the console from the ReactAPI warning engine at this point as well. That's ok; it's just telling us there are things to be aware of when working with the event. For now, we can ignore the warnings are recognize that this synthetic event object looks a little bit different than the original object that we would see with the DOM. The key takeaway here is that components can talk to buttons on event dispatch. From now on, this is how you should deal with events.

Apply What You've Learned

Now, let's build out a little Application that can handle some data that we pass through a few JSX elements.

We're going to build out some event handler functions for the following events across multiple elements:

```
-onClick - onDoubleClick - onMouseEnter - onChange;
```

Build out a `singleClickHandler` function that will call `alert("Single Click!");`

```
singleClickHandler = () => alert("Single Click!");
```

And add it to the first Button in our App.

```
<button onClick={this.singleClickHandler}>Click Handler Demo</button>
```

Lets do the same with a `doubleClickHandler` function, a `mouseEnterHandler` function and an `changeHandler` function.

```
doubleClickHandler = () => alert("Double Clicked!");

mouseEnterHandler = () => alert("Mouse Entered");

changeHandler = () => alert("Item was changed");


<div className="App">
  <h1>Hello Handlers</h1>
  <h2>Lets build out some handler functions.</h2>
  <button onClick={this.singleClickHandler}>Click Handler Demo</button>
  <button onDoubleClick={this.doubleClickHandler}>Double Click Handler</button>
  <div onMouseEnter={this.mouseEnterHandler}>Mouse Enter</div>
  <input onChange={this.changeHandler} placeholder="Change my input" />
</div>
```

Play around with the events on the screen and see how things are interacting one with another.

The input `changeHandler` for a min. Let's pass in the synthetic event through the function body. One thing that is cool about an input field is that we can pull in the resources it uses off of the `event.target.value` property that is found on the Synthetic Event object that will be passed in.

```
changeHandler = (event) => alert(event.target.value);
```

So now, as we type, we can see what's in the input field. Let's take this a step further and set up a state object. Then use a new function `setState` to reset the state on that object. Add this `constructor to our class Component` .

```
  constructor() {
    super();
    this.state = {
      someValue: '',
    }
  }
```

Now change our change handler to look like this.

```
changeHandler = (event) => {
  this.setState({ someValue: event.target.value });
};
```

Great, setState will update our `someValue` property on our state object by simply typing in the input field. Let's prove this by logging our state object inside the render function.

```
render() {
  console.log(this.state);
...
```

Check your console!

https://codesandbox.io/s/rmnj2r1o0p Here is a working copy of what we achieved!

Challenge

Using this Code sandbox that we've been working in together, build out an app that holds a string item on the state object.

- Build out a function handler that can take in the edits from an `onChange event` and then use the `setState` function to set those items on state.
- You should have seen me do this already, so you may have to refer back to previous walk-throughs.

Projects

- **React Todo**
- **React-Sorting-Hat**

  Stretch assignment for those that have finished the Todo Project beyond MVP

Review

Class Recordings

You can use class recordings to help you master the material.

- Class Components for Web35 w/ Warren Longmire
  Class Components
- All previous recordings

Demonstrate Mastery

To demonstrate mastery of this module, you need to complete and pass a code review on each of the following:

- Objective challenge:
  Let's take the functionality of this class component that we built earlier and extend it just a little bit.
  Declare a `Functional Component` called `RenderMessage` inside this CodeSandbox.
  - Make sure you declare your Props Object that will be passed into this component.
  - Return a `div` who's child is `props.message`
  - Now inside of the `App` class pass in that `RenderMessage` component and pass down a message prop to `RenderMessage` . This message prop should be set equal to the message property on the state object.
  - Once it's all wired up properly you've done it!
- Objective challenge:
  Using the following tools:
  - Class component
  - functional FormComponent, MessageComponent
  - click, and change handlers
  - `setState`

  Build out a form that will allow a user to handle data. You'll need a button, input field, and some data-bound to a DOM element that displays what the user is submitting.
  When a user clicks submit, show the data that's on state in an `alert` statement.
  Stretch Loop over a list of items showing those items to the screen. (Can be a list of strings).
  When a user clicks submit, instead of logging the item, push an item into that list, and watch the magic happen.
- Objective challenge:
  Using this Code sandbox that we've been working in together, build out an app that holds a string item on the state object.
  - Build out a function handler that can take in the edits from an `onChange event` and then use the `setState` function to set those items on state.
  - You should have seen me do this already, so you may have to refer back to previous walk-throughs.