LO6. Introduction to REACT

[Introduction and Installation 1](#_Toc97504004)

[First REACT Code 2](#_Toc97504005)

[REACT Components 4](#_Toc97504006)

[Stateful REACT 5](#_Toc97504007)

[Prop Validation 8](#_Toc97504008)

Introduction and Installation

#### What is REACT?

REACT is a nodejs library used to build front-ends for nodejs web applications. The library was developed by Facebook and is used to develop the view layer of a web app. REACT uses a component model and so therefore we create components which, in turn, can be reused. This is one reason for REACT being one of the most popular js libraries.

#### Installation

Installing REACT can be done simply by creating a template REACT app using the following command:

npx create-react-app \_\_\_\_\_

where you can specify an app name in the blank. Note that app names cannot contain capital letters and only hyphens are allowed aside from alphanumeric characters. This process will take some time to run as node will have to download the package content which is quite a bit larger than other packages. Once this is done, we will have a react app residing in the directory named for the app we created. If you received an error at this point, see below "Troubleshooting Install". Navigate into that directory and type

npm start

We can do this whenever we want and leave it. npm will monitor files and if we change anything, automatically restart our app for us. If everything worked correctly, the app will start and open a browser window. In the browser window, you should see a rotating REACT logo.

Now to look at some of the files created. You will find most files in two directories: public and src. First navigate to the public directory and open the index.html file. DO NOT CHANGE THIS FILE. You can scroll to the body of the file and note a div with an id set to 'root'. This is where our app will eventually be rendered.

We will be creating our first real REACT app in these directories so we can delete most of the files. In the src directory, delete everything. You can choose to leave the public directory alone or you can delete everything except the index.html file. The main thing is that we clear out the src file since that is where all of the files to be created will reside.

That's initial setup done. Now to start writing some REACT code.

**Troubleshooting Install**

One issue that can happen in Windows environments is if your user directory is your first and last name with a space. npx doesn't allow this. Part of the initial setup is to install REACT which it will attempt to do in the npm cache directory. The default for this is : "C:\Users\<USERNAME>\AppData\Roaming\npm-cache". If this is the case for you, you will need to set a different cache directory. You can do this with the following steps:

1. edit the npm config file. At the command line:    npm config edit
2. look for a line that looks like cache=C:\Users\<USERNAME>\AppData\Roaming\npm-cache  
     
   it should be only a few lines down. Create a new folder, say, C:\npmcache and set that line, without the **;**, and with cache set to this folder.

Then try running the initial npx command again.

First REACT Code

We'll start by creating a file in the src folder called index.js. This is the main starting point for our app and is hardcoded into the app's config files.

Inside this file, add the following lines:

import React from 'react';  
 import ReactDOM from 'react-dom';

We'll come back to this file once there is something to render to our browser window.

Now create a file in src called App.jsx. This is an extensible form of javascript where jsx stands for JavaScript XML. Essentially it allows us to write html directly into our javascript. We'll start by adding an import and creating our first component.

Add the import line:  import React, {Component} from 'react';

Now add the following after this:

class App extends Component {  
  
}

Now that we have a component called App, we need to be able to have it return html to be rendered. This is done by adding a render() method as follows:

class App extends Component {  
     render(){  
 return(   
  
 );  
     }  
}

One of the most common mistakes at this point is getting mixed up regarding parentheses and brackets. Add something simple to display:

<h1>My Header</h1>

inside the return block. Finally, we have to add an export line at the end of the file to export our component to be used:

export default App;

Now return to the index.js file and add the following lines after the import lines:

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

This imports the component into this index.js file and then renders it to the browser window, within the div we saw in the index.html file(the one with id='root').

Once both files are saved, npm will restart the app and you should see the following in the browser window:

Graphical user interface, text, application, email

Description automatically generated

Congratulations! You've created your first REACT component!

REACT Components

REACT is heavily component-based. This means we can not only create properly encapsulated components, we can embed them within other components. Create two additional components in the same App.jsx file. We'll call them Header and Content.

class Header extends Component {  
 render(){  
 return(  
 <h1>My Header</h1>  
 );  
 }  
}  
  
class Content extends Component {  
 render(){  
 return(  
 <p>My Content</p>  
 );  
 }  
}

We can then embed these components within our original App component. We do this by referencing them as xml tags:

class App extends React.Component {  
 render(){  
 return(  
 <div>  
 <Header />  
 <Content />  
 </div>  
 );  
 }  
}

Note that in this case, Header and Content must be enclosed in a div. When there is only one thing, this isn't required but when there are two or more adjacent components, tags, etc they have to be enclosed in a container of some kind.

Since these are classes, we can also invoke a constructor method right before render(). This is done if we want to include state within our component and is discussed in the next item.

Stateful REACT

## State

Maintaining state within a REACT component requires the use of a constructor method and the this.state object. For our initial App component this would be:

class App extends Component {  
 **constructor(){**  
 **super();**  
 **this.state = { ... }**  
 **}**  
 render(){  
 return(  
 <h1>My Header</h1>  
 );  
 }  
}

Note that we also invoke the parent constructor with super(). this.state will contain any data we wish to have available to work with in our component. It can contain one or more items and the items can be different data types or arrays. Let's update our App's state by adding an array of objects called data.

constructor() {  
 super();  
 this.state = {  
 data: [ { },{ },{ } ]  
 }  
}

now some fields('id', 'name' and 'age') and data in those objects:

constructor() {  
 super();  
 this.state = {  
 data:   
 [  
 {  
 "id":1,  
 "name":"Foo",  
 "age":"20"  
 },  
 {  
 "id":2,  
 "name":"Bar",  
 "age":"30"  
 },  
 {  
 "id":3,  
 "name":"Baz",  
 "age":"40"  
 }  
 ]  
 }  
}

We could also have this along with a single valued variable as well:

constructor() {  
 super();  
 this.state = {  
 **myvar: "This is just some text",**  
 data:   
 [  
 ...  
 ]  
 }  
}

So the built-in state object allows us to store whatever data we need within our component.

We may also need some way to get data into a component as well. This can be done using Properties.

### Props

Props or properties are immutable variables which allow a means to pass data into a component. Properties are named by the passing module and that naming is then used to reference the property data in the receiving component.

There are three ways to look at props:

1. Passing properties into our top level component from index.js
2. Passing properties to child components
3. Using default properties

Properties can be passed into our top level component by altering the ReactDOM.render as follows:

ReactDOM.render(  
 <App headerProp = "Header from props..." contentProp = "Content from props..." /> ,   
 document.getElementById('root'));

These can then be referred to in our App component via the **this.props** object after including the parameter **props** in the constructors.

constructor(props){  
 super(props);  
  
}  
render(){  
 return(  
 <div>  
 <h1>{this.props.headerProp}</h1>  
 <h2>{this.props.contentProp}</h2>  
 </div>  
 );  
}

We can also use default properties by adding the following:

App.defaultProps = {  
 headerProp: "Header from default props...",   
 contentProp: "Content from default props..."  
}

Finally, we can use props as a mechanism to pass state data from the parent component to child components:

Consider our component with this.state containing an array of objects called data as shown above. We could create a table in the parent object and then create a table row object as a child and pass our data array to the child to be outputted in table rows.

The table row component could be created as:

class TableRow extends Component {  
 render(){  
 return(  
 <tr>  
 <td>{this.props.tableData.id}</td>  
 <td>{this.props.tableData.name}</td>  
 <td>{this.props.tableData.age}</td>  
 </tr>  
 );  
 }  
}

Then the component can be embedded within a table in the App class using the array map function:

<table>  
 <tbody>  
 {this.state.data.map((person, i) => <TableRow key = {i} tableData = {person} />)}  
 </tbody>  
</table>

Here we use an arrow function (person, i) => <TableRow key = {i} tableData = {person} />

where person is the element to be mapped, i is the index and TableRow is the receiving component which will have two variables, i and person, passed as properties key and tableData to be referenced within the TableRow component. Note that 'tableData' is just an identifier and this name is not dependent on any other identifier used in this.state.

Prop Validation

Recall that javascript is weakly typed so the values of the properties we set can be any type. It may be useful to perform some type validation on props which can be done using a package called Prop-Types.

First, start by importing from the prop-types package:

import PropTypes from 'prop-types';

Then validation can be set up for the component by creating \_\_\_.propTypes to define the validation rules we wish to use. In our case, we will create App.propTypes.

App.propTypes = {  
 propString: PropTypes.string,  
 propNumber: PropTypes.number,  
 propBoolean: PropTypes.bool.isRequired,  
 propObject: PropTypes.object,  
 propArray: PropTypes.array.isRequired,  
 propFunction: PropTypes.func  
}

We can set up some sample data with default props:

App.defaultProps = {  
 propString: "Some string",  
 propNumber: 10,  
 propBoolean: true,  
 propObject: {  
 field1:"Val1",  
 field2:"Val2"  
 },  
 propArray: [1,2,3,4,5],  
 propFunction: function(x){return x}  
}

Finally, we can add some content to the component to display the props.

<div>  
 <p>String: {this.props.propString}</p>  
  <p>Number: {this.props.propNumber}</p>  
  <p>Bool: {this.props.propBoolean ? "True..." : "False..."}</p>  
  <p>Object: {this.props.propObject.field1}</p>  
  <p>Object: {this.props.propObject.field2}</p>  
 <p>Array: {this.props.propArray}</p>  
  <p>Func: {this.props.propFunction(3)}</p>  
</div>

To check the typing, try changing one of the default prop values to an invalid type and then check the console.  Any exceptions raised will show in the console.