WA2583 React JavaScript Programming

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Setting Up a React Development Environment

In this lab you will create a development setup that supports React development. The setup will take care of three essential issues:

- Transpiling ES6 code
- Transpiling JSX code
- Serving React Application files (index.html, etc.)

Node.js and npm should have been installed on your machine as part of the course setup. You will need them to complete this lab.

Part 1 - ES6 Transpilation support

In this part you will setup a development environment that transpiles ES6.

__1. Open a command prompt

__2. Create a directory to work in named:

```
C:\LabWork
```

__3. Create a directory "react-dev-env" inside the LabWork directory:

```
C:\LabWork\react-dev-env
```

This directory will be referred to in the lab as the **project root**.

__4. Navigate into the directory you just created

__5. Create a new package.json file using the following command (take defaults for all settings):

```
npm init
```

__6. Use node package manager to install each of the following *development* dependencies into the project:

```
babel-cli
babel-preset-es2015
chokidar
```

Hint: `npm install package-name --save-dev`

es2015 is another name for JavaScript ES6

chokidar helps babel watch for file changes

__7. Add the following scripts to the package.json file

```
"babel": "babel src --out-dir dist",
"babelw": "babel src --watch --out-dir dist"
```

__8. Create a ".babelrc" file in the project root with the following contents:

```
{"presets": ["es2015"]}
```

__9. Create a "\src" directory in the project directory

__10. Create a file named "test.js" in the "\src" directory.

__11. Add some ES6 code to test.js file. The code below can be used, or you can add your own. When done save the file.

```
class myclass{
  constructor(name){this.name = name;}
  display(){console.log(this.name);}
}
new myclass("myclass").display();
```

__12. Run the babel compiler using one of the scripts you added to package.json:

```
npm run babel
```

__13. Check that babel did it job (test.js should have been created in dist dir)

__14. Open \dist\test.js in and editor. The code you see should look like this:

```
var _createClass = function () { ... }();

function _classCallCheck(instance, Constructor) { if (!
(instance instanceof Constructor)) { throw new
TypeError("Cannot call a class as a function"); } }

var myclass = function () {
 function myclass(name) {
    _classCallCheck(this, myclass);

    this.name = name;
  }

  _createClass(myclass, [{
    key: "display",
    value: function display() {
```

```
      console.log(this.name);
    }
  }]);

  return myclass;
}();

new myclass("myclass").display();
```

This is the result of the ES6 transpilation.

__15. Use node.js to run the test file:

```
node dist\test.js
```

__16. The output should be:

```
myclass
```

__17. The current setup is suitable for development of standalone JavaScript code that uses ES6 and needs to be compiled to ES5. In the next part we will add JSX transpilation.

Part 2 - JSX Transpilation support

In this part you will add JSX transpilation support to the development environment.

__1. Use node package manager to install the following dependencies into the project:

react@15.6.2

react-dom@15.6.2

prop-types@15.6.0

```
Hint: npm install package-name --save
      for package-name use name@version_number above
```

__2. Use node package manager to install the following *development* dependencies into the project:

babel-preset-react

```
Hint: npm install package-name --save-dev
```

__3. Add "react" to the ".babelrc" file as shown here:

```
{"presets": ["es2015","react"]}
```

__4. Create a basic index.html file in the project root with the following contents:

```
<!doctype html>
<html>
<head>
<title>My React Project</title>
</head>
<body>
</body>
</html>
```

__5. Modify the index.html file:

- Add <script> tags to the <head> section to include the react and react-dom JavaScript libraries.
- Add a to the body with "id=main"
- Add a <script> tag just before the ending </body> tag to include an 'app.js' file from the "dist" directory.
- Save index.html

__6. When you are done your index.html file should look something like this:

```
<!doctype html>
<html>
<head>
<title>My React Project</title>
<script src="node_modules/react/dist/react.js"></script>
<script src="node_modules/react-dom/dist/react-
dom.js"></script>
</head>
<body>
<div id="main"></div>
<script src="dist/app.js"></script>
</body>
</html>
```

__7. Create an "app.js" file in the "\src" directory with the following contents:

```
ReactDOM.render(
    <h3>My App Title</h3>,
```

```
    document.getElementById('main'));
```

__8. Run the babel compiler:

```
npm run babel
```

Note: running "babelw" instead of "babel" will open it in "watch" mode. In "watch" mode babel will stay active after it runs and will run again every time you save changes to your files.

__9. Check that babel did its job. Open the transpiled file from the dist directory in an editor:

```
\dist\app.js
```

__10. You should notice that the content from the original file that appeared like this:

```
<h3>My App Title</h3>
```

Now appears like this in \dist\app.js

```
React.createElement(
 'h3',
  null,
 'My App Title'
)
```

This is the result of React JSX transpilation.

__11. Open index.html from the file system in the Chrome browser. The page should show "My App Title".

__12. Some application operations require the index.html to be served from a file server instead of directly from the file system. In the next part we will add a development server to the setup.

Part 3 - File Server Support

In this part you will add a development file server to the setup.

__1. Use node package manager to install the following *development* dependencies into the project:

http-server

```
Hint: npm install package-name --save-dev
```

__2. Add the following script to the package.json file

```
"start": "http-server"
```

__3. Start the server:

```
npm start
```

__4. Open index.html in the Chrome browser using the following url:

```
http://localhost:8080/index.html
```

__5. The page should load as before but this time from the development file server.

Part 4 - Review

In this lab you have created a development setup that:

- Transpiles ES6 code
- Transpiles JSX code
- Serves React Application files (index.html, etc.)

Lab 2 - Basic React Components

In this lab you will build a React application from functional components.

Part 1 - Create a Main App Components

In this part you will create a main App component that can hold other components.

__1. The following steps describe edits you need to make to the src\app.js file.

__2. Create a function named App() that uses JSX to return a component with the following structure:

```
<div><h3>My App Title</h3></div>
```

__3. Modify the ReactDOM.render function to render the App component you just created.

__4. When you are done with the above steps your src\app.js should look like this:

```
function App(){
    return <div><h3>My App Title</h3></div>;
}

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

__5. Add a functional component named Header that displays the app title.

__6. Modify the App function. Remove the <h3> tag and replace it with a JSX tag for the Header component you just created.

__7. The resulting src\app.js should look like this:

```
function App(){
    return <div><Header /></div>;
}

function Header(){
    return <h3>My App Title</h3>;
}

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

__8. Create a variable named "title" with the value "My React App"

__9. Replace the contents of the <h3> in the Header component with a JSX expression so

that it displays the "title" variable you just created.

__10. The changes made to src\app.js should look like this:

```
const title = "My React App"

function Header(){
    return <h3>{title}</h3>;
}
```

__11. Add a functional component named Body that displays a div containing a paragraph with some random text.

__12. Add the Body component to the JSX code on the App component.

__13. Add a functional component named Footer that displays a div containing an <h4> tag with the text "App Footer"

__14. Add the Footer component to the JSX code on the App component.

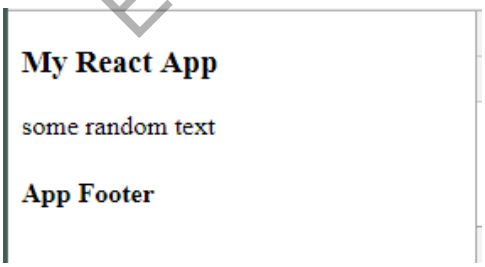
__15. When you are done the component functions should look something like this:

```
function App(){
    return <div><Header /><Body /><Footer /></div>;
}

function Body(){
    return ( <div><p>some random text</p></div> );
}

function Footer(){
    return ( <div><h4>App Footer</h4></div> );
}
```

And the browser window should look like this:



Part 2 - Pass Properties to Components

In this part you will create variables to hold property values and pass them to components.

__1. Create two variables as shown here:

```
var footerText = "footer text"
var author = { name:"John Doe",
               phone: "800-555-1212",
               email: "jdoe@gmail.com" }
```

__2. Pass footerText as a property named "text" to the Footer component where it appears inside the App component.

__3. The resulting App component should look like this:

```
function App(){
    return (
    <div>
        <Header />
        <Body />
        <Footer text={footerText} />
    </div> );
}
```

__4. Modify the Footer functional component to take "props" as a parameter and then use a JSX expression to replace the text in the <h4> tag with "props.text"

__5. The resulting Footer component should look like this:

```
function Footer(props){
    return <div><h4>{props.text}</h4></div>;
}
```

__6. Refresh the browser and make sure the footer is showing the property value that was passed to it.

__7. Lets do the same with the Body component

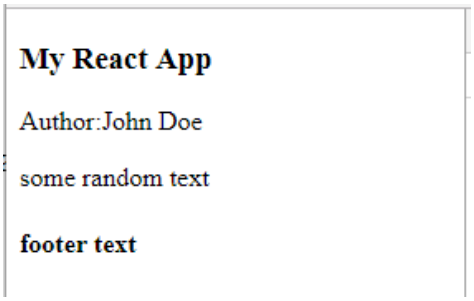
- Pass "author" as a property named "author" to the Body component.
- Modify the Body component to take "props" as a parameter.
- Add a paragraph before the existing paragraph in the Body component that displays the author's information.

__8. The resulting Body component might look like this:

```
function Body(props){
    return ( <div>
        <p>Author:{props.author.name}</p>
        <p>some random text</p>
```

```
            </div> );
    }
```

__9. The corresponding browser output would look like this:



Part 3 - Add some style

In this part we will add some CSS styles to the components.

__1. Edit Index.html

__2. Add a <style> section

__3. In the <style> section add section for a class named "boxed" with the following rules:

- Width = 200px (width: 200px;)
- Solid single line black border (border: 1px solid black;)

__4. Edit src\app.js

__5. Modify the App component.

- add the "boxed"class to the outer div of the component
- the syntax required for this is specified in JSX as

```
<div className={'boxed'}>
```

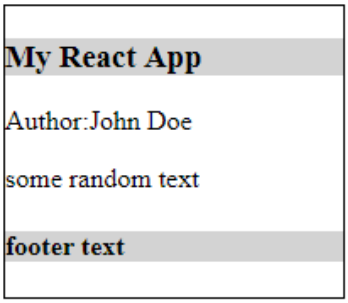
__6. Add the following style directly to the <h3>, <h4> elements of the Header and Footer components respectively.

```
backgroundColor: lightgrey
```

__7. The code to do this in JSX would look something like this

```
<h3 style={{backgroundColor: 'lightgrey'}} >
```

__8. The resulting browser output will look like this:



__9. This does not look so great. Lets add the following also:

```
margin: '0px',
padding: '5px',
textAlign: 'center',
```

__10. We probably don't want to place these all inline in the component. Instead lets create a variable to hold them:

```
const divStyle = {
    backgroundColor: 'lightgrey',
    margin: '0px',
    padding: '5px',
    textAlign: 'center',
};
```

__11. Now we can use the variable in the component like this:

```
<h3 style={divStyle} >
```

__12. The final Header and Footer components would look like this:

```
function Header(){
    return <h3 style={divStyle} >{title}</h3>;
}

function Footer(props){
    return ( <div>
```

```
                    <h4 style={divStyle} >{props.text}</h4>

                                </div> );

        }
```

__13. The browser output should now look like this:

Part 4 - Review

In this lab we have:

- build a React application from functional components
- passed parameters to components
- set styles and classes in components

Lab 3 - More React Component Concepts

In this lab you will work with the following React component concepts:

- Moving state into the root component
- Displaying Lists
- Updating input fields
- Click Event handling
- Passing parameters to event handlers

This lab picks up where the previous lab left off. If you did not do the previous lab you can use the solution file from that lab to get started. Solution files are available in the C:\LabFiles\Solution directory.

Part 1 - Moving state into the root component

In this part we will move some state into the root component. The root component in our case is the App component. We'll start by moving the title property.

__1. The title property is currently displayed in the Header component. The variable "title" is in scope so it is used directly.

```
const title = "My React App"

function Header(){
    return <h3 style={divStyle} >{title}</h3>;
}
```

__2. Instead of relying on scope lets pass the title in as a property like we did with the "text" in the footer.

- Edit the Header component
- Pass "props" into the Header component function
- Change the JSX expression that displays the title to get its value from "props.title"
- Update the App component to pass title as a property to the Header component

The resulting Header component should look like this:

```
function Header(props){
    return <h3>{props.title}</h3>;
```

```
}
```

And the Header tag in the App component looks like this:

```
<Header title={props.title}/>
```

__3. The App component currently looks like this. Notice how the values for title, author and footerText are coming from variables defined outside the App component. This works because they are in scope but we'd like to move those values inside the component.

```
function App(){
    return (
    <div className={'boxed'}>
        <Header title={title}/>
        <Body author={author} />
        <Footer text={footerText} />
    </div> );
}
```

__4. Setup the App component to take a single object containing all the state values.

- Add "props" as a parameter to the App component method.
- Combine title, author and footerText into a single object where they exist as properties. Name the object "scope"
- Use the spread operator to pass the scope to the App component when it is rendered.

__5. The resulting changes should look like this:

```
var scope = {
    title: "My React App",
    footerText: "footer text",
    author: {
            name:"John Doe",
            phone: "800-555-1212",
            email: "jdoe@gmail.com"
        }
    }

function App(props){
    return (
    <div className={'boxed'}>
        <Header title={props.title}/>
```

```
            <Body author={props.author} />
            <Footer text={props.footerText} />
        </div> );
}

ReactDOM.render( <App {...scope}/>,
    document.getElementById('main')
);
```

__6. Test to make sure the app is still working.

Part 2 - Displaying Lists

Displaying lists is essential to creating real applications. In this part we will display a list of books in our app.

__1. Add "books" to the scope object

```
books: [
    {isbn:'123', title:'The Time Machine', price:5.95 },
    {isbn:'123', title:'War of the Worlds', price:6.95 },
    {isbn:'123', title:'The Invisible Man', price:4.95 }
];
```

__2. Add a BookList functional component to the Body component. BookList should takes props as a parameter and returns an unordered list of book items

- Create the BookList component
- Edit BookList to return
- Inside the <ul> tags add a JSX expression that uses the array.map function to display a list of book titles
- Add BookList right after the author paragraph in the Body component.
- Make sure to pass books to the Body component and inside the Body component pass books again to BookList

__3. The resulting code changes should look like this:

```
var scope = {
    title: "My React App",
```

```
    footerText: "footer text",
    author: {
            name:"John Doe",
            phone: "800-555-1212",
            email: "jdoe@gmail.com"
        },
    books: [
        {isbn:'123', title:'The Time Machine', price:5.95 },
        {isbn:'456', title:'War of the Worlds', price:6.95 },
        {isbn:'789', title:'The Invisible Man', price:4.95 }
        ]
    }

function BookList(props){
    return ( <ul>
        {props.books.map(
            (book, index) => {return (
            <li key={index} >{book.title}</li> )}
        )}
        </ul>
    );
}

function Body(props){
    return ( <div>
        <p>Author:{props.author.name}</p>
        <BookList books={props.books} />
        <p>some random text</p>
        </div> );
}

The Body tag in App() should look like this:
<Body {...props} />
```

__4. The browser output should look like this:

Part 3 - Updating input fields

In this part we add an <input> field and update the app so that when users type into the field the changes are shown on the screen.

__1. Add a "color" property to the scope object with the value "blue"

__2. Add an input field just before the tag in the Body component

- The type of the input should be "text"
- The value of the field should come from the "color" property of the scope object
- The name of the input field should be set to 'color'
- You will need to pass the "color" property to the Body component where it appears in the App component. You can do this the same way you passed author and books or you can replace them with the spread operator {...props} which will then create properties inside of Body for all the properties in scope.

__3. Modify the text in the <p> above the input field. Change it from "some random text" to "Enter your favorite color:"

__4. The App and Body components should now look like this:

```
function App(props){
    return (
    <div className={'boxed'}>
        <Header title={props.title}/>
        <Body {...props} />
        <Footer text={props.footerText} />
    </div> );
}

function Body(props){
    return ( <div>
        <p>Author:{props.author.name}</p>
        <BookList books={props.books} />
        <p>Enter your favorite color:</p>
        <input type='text' name='color'
               value={props.color} />
        </div> );
}
```

__5. And the input field should be visible in the browser:

__6. Note though that if you try to enter any text into the field it does not appear on the screen. This is because React keeps supplying the same color value to the input field each time it is rendered. To fix this we need to respond to any changes the user makes to the input field and update the color value on the scope object.

__7. Before we do this we need to wrap the ReactDOM.render() call in a method so that we can call it after the "color" has been updated.

```
function renderApp(scope){
    ReactDOM.render( <App {...scope}/>,
        document.getElementById('main')
    );
}
```

__8. And since ReactDOM.render is no longer executing (since its wrapped inside a function now) we need to call this new function as the last line of the \src\app.js file:

```
renderApp(scope);
```

__9. Now add an onChange handler to the input field:

```
onChange = {handleChange}
```

__10. Next add a handleChange method to \src\app.js The method should:

- take "event" as a parameter
- update the "color" property of the scope object
- make a call to renderApp and pass the scope object

__11. The handleChange method should look like this:

```
function handleChange(event){
     scope[event.currentTarget.name] = event.currentTarget.value;
     renderApp(scope);
}
```

__12. Now text entered into the input field will be visible in the input field as you type it and it will be added at the same time to the scope.color property.

Part 4 - Click Event Handling

In the last part we saw how to handle the onChange event of input fields that are used in React components. In this part we will take a look at handling onClick events of buttons.

__1. Add an input element with type=button right below the other input field in the Body component.

- Input field's type is button
- Add an onClick handler named "handleButtonClick"
- Add a value attribute set to "Click Here"

__2. Add a "message" property to the scope object. For now set its value to an empty string.

__3. Add paragraph below the button that gets its text from the "scope.message" property.

__4. The Body component should now look like this:

```
function Body(props){
    return ( <div>
        <p>Author:{props.author.name}</p>
        <BookList books={props.books} />
        <p>Enter your favorite color:</p>
        <input type='text' name='color'
          onChange = {handleChange}
          value={props.color} />
        <input type='button' value="Click Here"
          onClick = {handleButtonClick} />
        <p>{props.message}</p>
        </div> );
}
```

__5. Add a "handleButtonClick" method to \src\app.js"

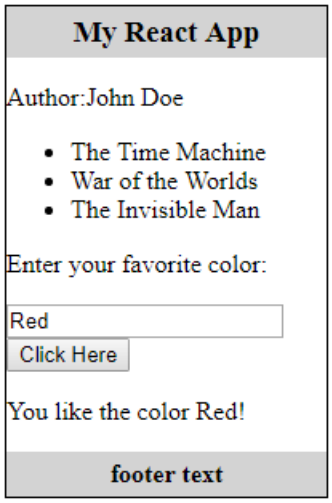
- code the handler to set the scope.message so that it shows like this, with the color

coming from scope.color:

```
You like the color Red!
```

- As the last line of the handler make a call to renderApp

__6. Try out the app. Refresh the browser, enter your favorite color and click the button. The message below the button should update to reflect your choice.



Part 5 - Passing Parameters to Event Handlers

In this part we will take a look at passing parameters to the onClick event handler for list items.

__1. Add a property named "selectedIndex" to the scope object. Set its value to -1.

__2. Add a method to \src\app.js named "handleListItemClick"

- The method should take two parameters, "event' & "index"
- It should use the "index" parameter to:
  - Set the "scope.selectedIndex" property
  - And to look up a book from the scope.booklist array.
  - After updating the selectedIndex the handler should make a call to renderApp.
  - As the last line of the method details of the chosen book should be output to the console.

-

__3. Add an onClick attribute to the <li> in the BookList component

```
onClick = {(e)=>handleListItemClick(e, index)}
```

We are using an arrow function which autobinds "this" and we are passing multiple parameters.

The event parameter is always passed first. After that you can pass whatever other parameters you need. Here we are passing the "index" parameter.

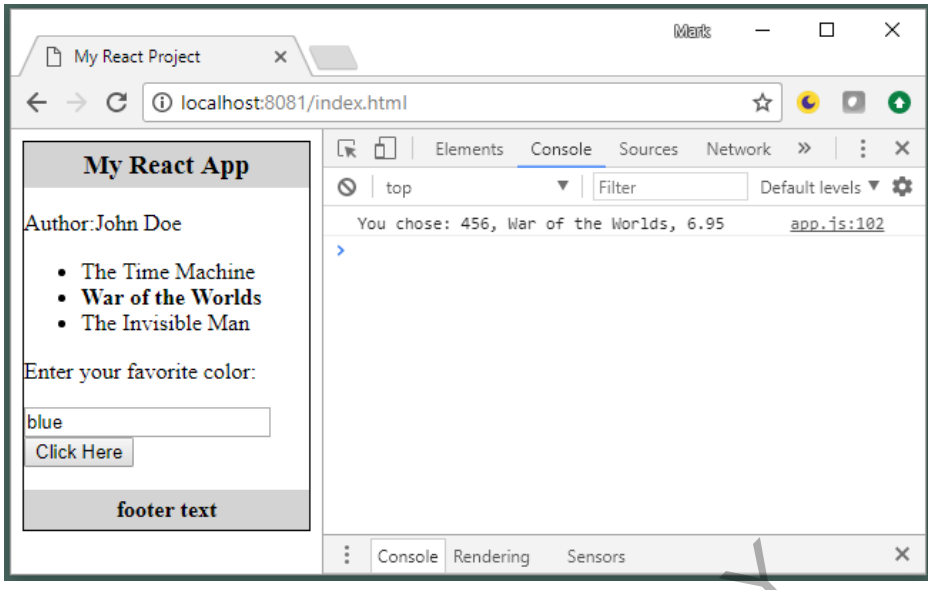
__4. Highlight the clicked item

- Edit index.html and add a style to the <style> section for a "selected" class that makes text bold when the class is applied
- Make sure to pass the selectedIndex to the BookList component
- Insert a statement into the <li> that sets the class to "selected" when the index of an item matches the selectedIndex

```
className={ index === props.selected ? "selected" : ""}
```

__5. Try testing the app.

- Open Chrome's Developer tools(F12) so you can see the JavaScript console.
- Refresh the app page in the browser
- Click on an item in the list
- The selected item should appear in bold
- The details of the item should appear printed out in the console



Part 6 - Review

In this lab we covered the following React component concepts:

- Moving state into the root component
- Displaying Lists
- Updating input fields
- Click Event handling
- Passing parameters to event handlers