Creating Single Page Applications with React.js

Introduction

In the previous assignment we introduced ***JavaScript*** as the language for programming the browser. We introduced ***jQuery*** as a popular JavaScript library that simplified common tasks such as binding to the DOM and interacting with the user. In this assignment we build on the JavaScript skills to introduce ***React.js***, a popular JavaScript library for building ***Single Page Applications*** (***SPAs***).

Labs

This section presents ***React.js*** examples to program the browser, interact with the user, and generate dynamic HTML. Use the same project you worked on last assignment. After you work through the examples you will apply the skills while creating a clone of ***Tuiter*** on your own. Using ***IntelliJ***, open the project you created in the previous assignment. From within IntelliJ, use ***File***, ***Open Project***, and navigate to the project directory, (***web-dev***), and click ***Open*** or ***OK***. ***Include all the work in the Labs section as part of your final deliverable***. TAs will grade the final result of having completed the whole ***Labs*** section.

Single Page Application

Single Page Applications (***SPAs***) render all their content dynamically into a single HTML document including navigation between various screens, without actually navigating away from the original HTML document. ***React.js*** achieves this by declaring a single HTML element where all the content will be render by the ***ReactDOM*** library. The code below declares a DIV with ID ***root*** where a ***React.js*** application will render all the content dynamically. Below ***div#root*** there's an example set of links to previous assignemnts. Update your ***public/index.html*** based on the example below.

```
<body>
  <div id="root"></div>
  <h1>Non React.js Assignments</h1>
  <ul>
    <li><a href="a2/index.html">Assignment 2</a></li>
    <li><a href="a3/index.html">Assignment 3</a></li>
    <li><a href="a4/index.html">Assignment 4</a></li>
    <li><a href="a5/index.html">Assignment 5</a></li>
  </ul>
</body>
```

The ***React.js*** application is implemented in ***src/index.js*** as shown below. The code imports the ***React*** and ***ReactDOM*** libraries.

```
import React from 'react';                  // core library
import ReactDOM from 'react-dom';           // library for rendering into DOM element
import App from './App';                    // application entry point, can be called anything, but App is common

ReactDOM.render(                            // use library to render
```

```
  <App />,                                  // App
  document.getElementById('root')           // into element whose ID is root, the one declared earlier in index.html
);
```

ReactDOM uses ***document.getElementById('root')*** to retrieve a reference to the ***DOM*** element declared in ***index.html***. ***ReactDOM*** then creates an instance of ***App*** and appends its output to the element whose ID is ***root***. The ***src/App.js*** is the entry point of the ***React.js*** application we're going to build and it might contain code generated by the ***create-react-app*** tool we used to create the project at the beginning of the course. Let's replace the content of ***src/App.js*** with the code below. It's basically a function called ***App*** that returns an ***H1*** element greeting the world. Note how the return statement is returning an ***HTML tag***, not an ***HTML string***. This is possible because ***React.js*** uses a library called ***JSX*** or ***JavaScript XML***. ***JSX*** allows mixing and matching ***JavaScript*** and ***XML*** seamlessly and ***HTML*** is just a particular flavor of ***XML***. This syntax greatly simplifies integrating HTML and JavaScript as if they were two sides of the same coin.

```
function App() {
  return (
     <h1>Hello World!</h1>
  );
}
export default App;
```

To test, start the React application using ***npm*** as shown below. Run the command from the root directory of your project.

```
npm start
```

Confirm the browser refreshes with the message shown. Alternatively create a ***Run Configuration*** by selecting ***Edit Configurations*** from the ***Run*** menu. In the ***Run/Debug Configuration*** dialog, click the plus sign and select ***npm***. In the ***Name*** field give a name such as ***run***. From the Scripts dropdown select start. Click ***Ok*** and confirm the configuration appears in the dropdown at the top right of the IDE, next to a green triangular play button. To run the React application select the run configuration and click on the green play button.

Bootstrap and Fontawesome

We're going to keep using the same styling libraries we've been using so far: Bootstrap and Fontawesome. Copy ***public/vendors*** directory to ***src/vendors*** so that the new React.js application can use the same styling libraries we've been using all along. To load the libraries from React.js code, you just have to import the files as shown in the code below. Note ***className*** instead of ***class*** since ***class*** is a reserved keyword in JavaScript. Confirm that the browser refreshes with Bootstrap styling.

```
import './vendors/bootstrap/css/bootstrap.min.css';
import './vendors/bootstrap/bootstrap.min.css';
import './vendors/fontawesome/css/all.min.css';

function App() {
```

```
  return (
    <div className="container">
      <h1>Hello World!</h1>
    </div>
  );
}
```

Hello World component

One of the strength of React.js is that it incourages breaking up large applications into smaller parts or ***components*** you can then assemble into sophisticated user interfaces. Let's create our first React.js component by breaking out the Hello World H1 element into a separate JavaScript file as shown below. In a new ***components/*** directory Create ***src/components/hello-world.js*** with the following content.

```
import React from "react";

const HelloWorld = () => {
   return(
      <h1>Hello World!</h1>
   )
};

export default HelloWorld;
```

We can then import thhe new component in ***src/App.js*** as shown below. Note the missing ***.js*** optional file extension in the ***HelloWorld*** import statement. Also note the new ***<HelloWorld/>*** tag matching the name of the import, file name, and function name.

```
import HelloWorld from "./components/hello-world";          // no .js extension needed

function App() {                                            // can also use const App = () => {
  return (
    <div className="container">
      <HelloWorld/>                                         // this calls HelloWorld() and appends return
    </div>
  );
}
```

Navigation

Earlier we mentioned that ***Single Page Application*** (***SPA***s) implement applications by dynamically rendering all content into a single HTML document and that we rarely or never navigate away from that one HTML document, so you might ask, how do we break up a large Website or application into several screens? The answer is that React.js can accomplish the same functionality by swapping different screens in and out of the single HTML document giving the illusion of navigating between multiple screens. Instead of building this feature ourselves from scratch, we'll use a popular navigation library called [React Router](). To practice navigating between various screens, lets create a couple of components that will serve as our screens we'll

develop further throughout the assignment. Under the ***components/*** directory, create components ***labs.js*** and ***tuiter.js***. They should both just return H1 elements with the name of the file as the header. Use the ***hello-world.js*** component as an example. Then import the new components and render them under the HelloWorld component as shown below. Refresh the browser and confirm that all three components are rendering as expected.

```
import HelloWorld from "./components/hello-world";      // imports hello-world.js
import Labs from "./components/labs";                   // imports labs.js
import Tuiter from "./components/tuiter";               // imports tuiter.js
function App() {
  return (
    <div className="container">
      <HelloWorld/>
      <Labs/>
      <Tuiter/>
    </div>
  );
}
```

Ok, now let's install the ***React Router*** library from the command line as shown below. Run the command from the root of the project. Note the ***--save*** option. Those are two dashes before the the "save". The option makes sure to add the library as a dependency in ***package.json***.

```
npm install react-router-dom --save
```

Once the library has fully downloaded and installed, let's add a browser router as shown below. The ***BrowserRouter*** tag sets up the base mechanism to navigate between multiple components. In this case we're going to navigate between the three components within the ***BrowserRouter*** tag, e.g., ***HelloWorld***, ***Labs*** and ***Tuiter***.

```
import {BrowserRouter} from "react-router-dom";
function App() {
  return (
    <BrowserRouter>
      <div className="container">
        <HelloWorld/>
        <Labs/>
        <Tuiter/>
      </div>
    </BrowserRouter>
  );
}
```

To navigate between components we use the ***Route*** component from ***React Router*** to declare ***paths*** and map them to corresponding component we want to render for that ***path***. Update your code as shown below.

```
import {BrowserRouter, Route, Routes}       // import Route
       from "react-router-dom";

function App() {
 return (
   <BrowserRouter>
     <div className="container">
       <Routes>
         <Route path="/hello"                // maps /hello to HelloWorld, which means
                element={<HelloWorld/>}/>    // HelloWorld component displayed if you
         <Route path="/labs"                 // navigate to http://localhost:3000/hello
                element={<Labs/>}/>
         <Route path="/tuiter"
                element={<Tuiter/>}/>
       </Routes>
     </div>
   </BrowserRouter>
 );
}
```

Having declared the routes, now the components won't all render at the same time in the same screen. Instead they will render when the URL in the browser matches the path declared in their parent Route. To test this, refresh your browser and navigate to http://localhost:3000/hello and confirm the ***Hellow World!*** message appears. Then confirm navigating to http://localhost:3000/labs displays ***Labs***. Then confirm navigating to http://localhost:3000/tuiter displays ***Tuiter***.

We can declare the ***Labs*** component as the default landing screen by mapping it to "/", the root path. Refresh the browser and confirm that ***Labs*** is now the default screen.

```
<BrowserRouter>
 <div className="container">
   <Routes>
     <Route path="/hello"
            exact={true}
            element={<HelloWorld/>}/>
     <Route path="/"
            exact={true}
            element={<Labs/>}/>
     <Route path="/tuiter"
            exact={true}
            element={<Tuiter/>}/>
   </Routes>
 </div>
</BrowserRouter>
```

Navigation links

Instead of typing the links in a browser's navigation bar, we can create hyperlinks in our components that navigate between components. The examples below implement navigation between all three components created so far. Refresh the browser and confirm you can navigate between all components.

hello-world.js	***labs.js***	***tuiter.js***

```
import React from "react";
import {Link} from "react-router-dom";

const HelloWorld = () => {
   return(
     <>
        <h1>Hello World!</h1>
        <Link to="/">
           Labs
        </Link> |
        <Link to="/tuiter">
           Tuiter
        </Link>
     </>
   )
};

export default HelloWorld;
```

```
import React from "react";
import {Link} from "react-router-dom";

const Labs = () => {
   return(
     <>
        <h1>Labs</h1>
        <Link to="/hello">
           Hello
        </Link> |
        <Link to="/tuiter">
           Tuiter
        </Link>
     </>
   )
};

export default Labs;
```

```
import React from "react";
import {Link} from "react-router-dom";

const Tuiter = () => {
   return(
     <>
        <h1>Tuiter</h1>
        <Link to="/hello">
           Hello
        </Link> |
        <Link to="/">
           Labs
        </Link>
     </>
   )
};

export default Tuiter;
```

Default index.js

The ***HelloWorld*** component is fairly simple, but the other two, ***Labs*** and ***Tuiter***, are going to get fairly complex throughout the assignment. Instead of having a single JavaScript document, let's create dedicated directories for each of the components so we can break out the work into separate efforts. Create a new folder ***src/components/labs***, move ***labs.js*** into the new directory, and rename the file to ***index.js***. Confirm that the application still renders fine. You might need to restart the application by typing ***Ctrl+C*** and then ***npm start*** again. The reason it still works is that the import in ***App.js*** refers to the component without the file extension, so it can refer to a corresponding file with the same name, or a folder with the same name, but containing an ***index.js*** file, the default file. Do the same for ***tuiter.js***. Create a new folder called ***tuiter***, move ***tuiter.js*** into the new folder and rename the file as ***index.js***. Confirm everything still renders the same.

Classes

Let's start practicing simple things, like classes and styles. Under the ***labs*** folder, create another folder called ***classes*** and create the following files under it.

index.js

```
import React from "react";
import './index.css';
const Classes = () => {
   return(
      <div>
         <h2>Classes</h2>
         <div className="wd-bg-yellow wd-fg-black wd-padding-10px">
            Yellow background</div>
         <div className="wd-bg-blue wd-fg-black wd-padding-10px">
            Blue background</div>
         <div className="wd-bg-red wd-fg-black wd-padding-10px">
            Red background</div>
```

index.css

```
.wd-bg-yellow {
   background-color: lightyellow;
}
.wd-bg-blue {
   background-color: lightblue;
}
.wd-bg-red {
   background-color: lightcoral;
}
.wd-fg-black {
   color: black;
}
```

```
      </div>
   )
};
export default Classes;
```

```
.wd-padding-10px {
   padding: 10px
}
```

From the ***labs*** component (***labs/index.js***) import the new ***Classes*** component (***classes/index.js***) as shown below. Confirm the new ***classes*** component renders in the ***labs*** screen. Confirm the component renders as expected.

```
import React from "react";
import Classes from "./classes";
const Labs = () => {
   return(
      <div>
         <h1>Labs</h1>
         <Classes/>
      </div>
   )
};
```

The previous example used static classes such as ***wd-bg-yellow***. Instead we could calculate the class we want to apply based on any convoluted logic. Here's an example of creating the classes dynamically by concatenating a ***color*** constant. Refresh the screen and confirm components render as expected.

```
const Classes = () => {
   const color = 'blue';
   return(
      <div>
         <h2>Classes</h2>
         <div className={`wd-bg-${color} wd-fg-black wd-padding-10px`}>
            Dynamic Blue background</div>
```

Even more interesting is using expressions to conditionally choose between a set of classes. The example below uses either a ***red*** or ***green*** background based on the ***dangerous*** constant. Try with ***dangerous true*** and ***false*** and confirm it renders red or green as expected.

index.js

```
   const color = 'blue';
   const dangerous = true;
   return(
      <div>
         <h2>Classes</h2>
         <div className={`${dangerous ? 'wd-bg-red' : 'wd-bg-green'}
                                 wd-fg-black wd-padding-10px`}>
            Dangerous background</div>
```

index.css

```
.wd-bg-green {
   background-color: lightgreen;
}
```

Styles

In HTML the ***styles*** attribute accepts a CSS string to style the element applied to. In React.js, the ***styles*** attribute does not accept a string, instead it accepts a JSON object where the properties are CSS properties and the values are CSS values. To practice how this works, under the ***labs*** directory, create a ***styles*** directory with the files below. The ***styles*** component (***styles/index.js***) declares constant JSON objects that can be applied to elements using the ***styles*** attribute. Alternatively, the styles attribute accepts a JSON literal object instance which results in a weird syntax of double curly brackets as shown below. Refresh the browser and confirm the browser renders as expected. Note we use ***background-color*** instead of ***backgroundColor***.

src/components/labs/styles/index.js

```
import React from "react";
const Styles = () => {
   const colorBlack = {
      color: "black"
   }
   const padding10px = {
      padding: "10px"
   }
   const bgBlue = {
      "backgroundColor": "lightblue",
      "color": "black",
      ...padding10px
   };
   const bgRed = {
      "backgroundColor": "lightcoral",
      ...colorBlack,
      ...padding10px
   };
   return(
      <div>
         <h1>Styles</h1>
         <div style={{"backgroundColor": "lightyellow",
            "color": "black", padding: "10px"}}>
            Yellow background</div>
         <div style={bgRed}>
            Red background</div>
         <div style={bgBlue}>
            Blue background</div>
      </div>
   );
};
export default Styles;
```

src/components/labs/index.js

```
import Styles from "./styles";
const Labs = () => {
   return(
      <div>
         <h1>Labs</h1>
         <Styles/>
         <Classes/>
      </div>
   )
};
```

Conditional output

Ok, enough styling. Let's play around with rendering content based on some logic. The following example decides to render one content versus another based on a simple boolean constant ***loggedIn***. If the user is

loggedIn, then the component renders a greeting, otherwise suggests the user should login. Implement the example in ***src/components/labs/conditional-output/conditional-output-if-else.js*** with the following code.

```
import React from "react";
const ConditionalOutputIfElse = () => {
   const loggedIn = true;
   if(loggedIn) {
      return(<h2>Welcome If Else</h2>);
   } else {
      return (<h2>Please login If Else</h2>);
   }
};
export default ConditionalOutputIfElse;
```

A more compact way to achieve the same thing is to include the conditional content in a boolean expression that shortcircuits the content if its false or evaluates the expression if it's true. Implement the equivalent component below in ***src/components/labs/conditional-output/conditional-output-inline.js***.

```
import React from "react";
const ConditionalOutputInline = () => {
   const loggedIn = false;
   return (
      <>
         { loggedIn && <h2>Welcome Inline</h2>}        // if loggedIn is true, then this expression evaluates
         {!loggedIn && <h2>Please login Inline</h2>}   // if loggedIn is false, then this expression evaluates
      </>
   );
};
export default ConditionalOutputInline;
```

Merge both components into a single component as shown below and then import the new component into the ***labs/index.js***. Confirm all components render as expected.

src/components/labs/conditional-output/index.js

```
import React from "react";
import ConditionalOutputIfElse from "./conditional-outputIfElse";
import ConditionalOutputInline from "./conditional-outputInline";
const ConditionalOutput = () => {
   return(
      <>
         <conditional-outputIfElse/>
         <conditional-outputInline/>
      </>
   )
};
export default ConditionalOutput;
```

src/components/labs/index.js

```
import conditional-output from
"./conditional-output";
const Labs = () => {
   return(
      <div>
         <h1>Labs</h1>
         <ConditionalOutput/>
         <Styles/>
         <Classes/>
      </div>
   )
};
```

ToDo List

In a previous assignment we created a ***Todo*** list application that rendered a list of todos dynamically using ***JavaScript*** and ***jQuery***. In this section we are going to port that work to React.js. You will then be able to tackle the ***Tuiter*** section later in the assignment.

Create a new directory in ***src/components/labs/todo***. Copy all the todo code you worked on in previous assignments into this new directory. You can skip the HTML files since we won't need them. Let's start porting the lowest level components first, the ones that have the least number of dependencies. Refactor ***todo-item.js*** as shown below and then import it in ***labs/index.js***. Confirm the new TodoItem component renders as before and the check box is initially selected.

Before	***After***	***Comments***

const TodoItem = (todo)	const TodoItem = ({
	todo = {
	done: true,
	title: 'Buy milk',
	status: 'COMPLETED'
	}
	})
		/*
		provide default object
		so we can test
		component standalone

=> { return(` <li> <input type="checkbox" ${todo.done ? 'checked' : ''}/> ${todo.title} (${todo.status}) </li> `); } export default TodoItem;	=> { return(<input type="checkbox" defaultChecked={todo.done}/> {todo.title} ({todo.status})); } export default TodoItem;	we don't need ` ` don't need $ use defaultChecked to set initial state don't need $ don't need ` ` */

Things to note when porting ***JavaScript*** functions to ***React.js*** components:

- Prefer object destructors as a single parameter instead of a list of parameters
- Provide default values for the parameters and/or parameter objects. This helps testing as a standalone components
- Remove or replace unnecessary string concatenation syntax such as back ticks
- Remove or refactor any use of jQuery symbols such as $
- Remove or refactor unnecessary string expressions such as ${}

Now refactor the ***todo-list.js*** as shown below.

Before	***After***	***Comments***
import TodoItem from "./todo-item.js"; import todos from "./todos.js"; const TodoList = () => { return(`	import TodoItem from "./todo-item"; import todos from "./todos.json"; const TodoList = () => { return(	/* .js is optional reformat as JSON don't need ` `

```
  <ul>
   ${
     todos.map(todo => {
       return(TodoItem(todo));
     }).join('')
     }
  </ul>
 `);
}
export default TodoList;
```

```
  <ul>
   {
     todos.map(todo => {
       return(<TodoItem todo={todo}/>);
     })
   }
  </ul>
 );
}
export default TodoList;
```

```
don't need $

use tag syntax instead
attribute as parameter
don't need join()

don't need ` `

*/
```

Additional things to note when porting JavaScript functions to React.js components:

- JavaScript file extensions are optional when importing them
- Data files should be JSON files formatted as such
- Replace nested function calls with equivalent tags
- Replace function parameter calls with attributes
- Remove string concatenation functions such as join('')

Refactor ***todos.js*** into a JSON file as shown below

```
[
  { "title": "Buy milk",          "status": "CANCELED",    "done": true   },   // properties must be double quoted
  { "title": "Pickup the kids",   "status": "IN PROGRESS", "done": false  },   // string values double quoted
  { "title": "Walk the dog",      "status": "DEFERRED",    "done": false  }    // boolean and numbers no quotes
]
```

Import ***todo-list.js*** in ***labs/index.js***, refresh the browser, and confirm the ***TodoList*** renders a list of checkboxes and todo items.

Tuiter

Now that you've had a chance to practice creating React.js components, let's apply these skills to port the ***Tuiter*** clone implementation from previous assignments. Create a new directory ***src/components/tuiter***, move ***tuiter.js*** into this new ***Tuiter*** directory, and rename it to ***index.js***. You might need to restart the application. Let's start with porting over components that have little dependencies on other components.

NavigationSidebar

The ***NavigationSidebar*** component created in a previous assignment is a good place to start because it has no dependencies on other components, so we should be able to port it easily and test it independently. Create ***src/components/tuiter*** directory and create all your new components in this new folder. Under the ***src/components/tuiter*** directory, create folder ***NavigationSidebar*** and copy the ***index.js*** from the previous assignment into this new directory. We will walk you step by step on how to port the first couple of components. As you gain confidence on how to do this, we will progressively give you less detailed instructions and let you do it on your own.

Import React.js

The first thing you need to do is to import the React.js library. All React components must at least import this one library. Open the ***index.js*** and import ***React.js***.

```
import React from "react";
const NavigationSidebar = ( ... ) => { ... }
```

Object deconstructed parameter

Convert all parameters into an object deconstructor and provide initial default values.

Before	***After***

const NavigationSidebar = (active) => { ... }	const NavigationSidebar = (
	  {
	  active = 'explore'
	  }) => { ... }

React function components return HTML elements, not strings

Previous components were rendered with string manipulation. Using JSX and React.js we can instead render HTML elements. Remove the backticks (`) quotations in the ***return*** statement.

Before	***After***
return(`   ... `);	return( );

React function components return a single HTML element

React.js function components can only return a single HTML element. If the function needs to return more than one element, then they need to be wrapped with a parent element. Most common element is a simple DIV. Alternatively you can use fragment syntax which looks like tags with no names <></>

Wrap with generic anonymous element	***OR wrap with DIV element***

return(
  <>
  ...
  </>
);
	return(
	
	  ...
	
	);

Replace class with className

The HTML ***class*** attribute is commonly used to associate CSS transformation rules to an HTML element. In JavaScript ***class*** is a keyword so we can't use it. Instead we use ***className*** which will be transpiled into ***class*** in the resulting DOM. Replace all ***class*** attributes with ***className***.

Before	***After***
   <a class="list-group-item"	 <a className="list-group-item"

Embedded expressions

If an attribute value is dynamic then refactor it to use string template expressions instead as shown below.

Before	***After***
<a class="list-group-item ${active === 'home' ? 'active' : ''}"	<a className={`list-group-item   ${active === 'home' ? 'active' : ''}`}

To test the ***NavigationSidebar*** component, import it from ***tuiter.js*** and pass it "***home***" for the ***active*** attribute. Refresh the browser and confirm the sidebar displays as expected.

```
import React from "react";
import NavigationSidebar from "./navigation-sidebar";
const Tuiter = () => {
  return(
    <NavigationSidebar active="home"/>
  )
};
export default Tuiter;
```

WhoToFollowListItem

The ***WhoToFollowListItem*** component created in a previous assignment is a good place to continue because it has no dependencies on other components, so we should be able to port it easily and test it independently. In the ***src/components/tuiter*** directory create folder ***WhoToFollowList*** and copy the ***who-to-follow-list-item.js*** from the previous assignment into this new directory. Apply the same strategies we used to port the ***NavigationSidebar*** component. Additionally remove unnecessary ***$*** symbols.

Before	***After***
	

Remove $ symbols from inline expressions

Before	***After***
@${who.handle}	@{who.handle}

Convert the parameter list into an object destructor expression and provide default values as shown below. This will allow creating an instance of the component for testing.

Before	***After***

const WhoToFollowListItem = (who) => { ... }	const WhoToFollowListItem = (
	  {
	  who = {
	  avatarIcon: '../../../images/nasa.png',
	  userName: 'NASA',
	  handle: 'NASA',
	  }
	  }) => { … }

To test the ***WhoToFollowListItem*** component, import it from ***tuiter.js***. You'll need to wrap the content with a DIV or fragment tag as shown below since components can only return a single element. Refresh the browser and confirm the ***NavigationSidebar*** and ***WhoToFollowListItem*** display as expected.

```
import React from "react";
import NavigationSidebar from "./navigation-sidebar";
import WhoToFollowListItem from "./who-to-follow-list/who-to-follow-list-item";
const Tuiter = () => {
  return(
    <>
      <NavigationSidebar active="home"/>
      <WhoToFollowListItem/>
    </>
  )
};
export default Tuiter;
```

Try passing a different value for the ***who*** attribute as shown below. Refresh the browser and confirm the ***WhoToFollowListItem*** displays as expected.

```
import React from "react";
import NavigationSidebar from "./navigation-sidebar";
import WhoToFollowListItem from "./who-to-follow-list/who-to-follow-list-item";
const Tuiter = () => {
  return(
    <>
      <NavigationSidebar active="home"/>
      <WhoToFollowListItem who={{
        avatarIcon: '../../../images/virgin.png',
        userName: 'Virgin Galactic',
        handle: 'virgingalactic',
```

```
        }}/>
      </>
    )
};
export default Tuiter;
```

WhoToFollowList

The ***WhoToFollowList*** component built in a previous assignment depended on the ***WhoToFollowListItem*** created in the previous section. The ***WhoToFollowList*** component loads an array of who objects and renders one item for each object in the array. In the ***src/components/tuiter/who-to-follow-list*** copy the ***index.js*** from the previous assignment into this new directory. Apply the same strategies we've presented so far to convert the code into a ***React.js*** component.

JavaScript extension is optional

The ***WhoToFollowList*** component imports the ***WhoToFollowListItem*** component by referring to the JavaScript file where it is implemented. In ***React.js*** and JavaScript modules in general, the ***.js*** file extension is optional and it is best practice to remove it when importing a module as shown below.

Before

```
import React from "react";
import WhoToFollowListItem from
       "./who-to-follow-list-item.js";
```

After

```
import React from "react";
import WhoToFollowListItem from
       "./who-to-follow-list-item";
```

Loading JSON files

The ***WhoToFollowList*** component imports a JavaScript file called ***who.js*** that declares an array of who objects representing people to follow as shown below on the left. In React.js we would instead use a JSON file as shown below on the right. To convert one to the other, rename the file with a ***.json*** extension. Remove the export default statements since this is not a variable. Also remove the trailing colons and commas. Finally, all attributes and string values must be ***double quoted*** as shown below.

Before: who.js

```
export default [
  {
    avatarIcon: '../../../images/java.png',
    userName: 'Java',
    handle: 'Java',
  },
  {
    avatarIcon: '../../../images/relativity.jpeg',
    userName: 'Relativity Space',
    handle: 'relativityspace',
  },
 ...
```

After: who.json

```
[
  {
    "avatarIcon": "../../../images/java.png",
    "userName": "Java",
    "handle": "Java"
  },
  {
    "avatarIcon": "../../../images/relativity.jpeg",
    "userName": "Relativity Space",
    "handle": "relativityspace"
  },
 ...
```

```
];
```

```
]
```

Mapping over an array

The ***WhoToFollowList*** component iterates over the ***who*** array using a ***map*** function. The ***map*** function evaluates the ***WhoToFollowListItem*** function for each element and collates the results into an array of strings. These are then joined into a single HTML string. In React.js we are not working with strings and instead collating component instances to form DOM elements. The code below demonstrates how to reimplement the ***jQuery*** implementation into a React.js implementation. Remove the unnecessary ***$*** and ***join()*** functions. Replace the ***WhoToFollowListItem*** function call with its tag representation. Pass the who instance as an attribute value. Refresh the browser and confirm the ***WhoToFollowListItem*** component renders as expected.

Before

```
${
   who.map(who => {
      return(
         WhoToFollowListItem(who)
      );
   }).join('')
}
```

After

```
{
   who.map(who => {
      return(
         <WhoToFollowListItem who={who}/>
      );
   })
}
```

PostSummaryItem

The ***PostSummaryItem*** is very similar to the ***WhoToFollowListItem***. Convert ***PostSummaryItem*** into a React.js component by applying the same strategies applied to convert ***WhoToFollowListItem*** into a React.js component. In directory ***src/components/tuiter/post-summary-list***, copy the ***post-summary-item.js*** from the previous assignment and convert it into a React.js compoment. Convert the parameter list into an object destructor expression and provide default values as shown below. This will allow creating an instance of the component for testing.

Before

```
const PostSummaryItem = (post) => { ... }
```

After

```
const PostSummaryItem = (
   {
      post = {
         "topic": "Web Development",
         "userName": "ReactJS",
         "time": "2h",
         "title": "React.js is a component based front end library
that makes it very easy to build Single Page Applications or
SPAs",
         "image": "../../../images/react-blue.png"
      }
   }) => {
```

To test the ***PostSummaryItem*** component, import it from ***tuiter.js***. Refresh the browser and confirm the ***PostSummaryItem*** displays as expected. Try passing a different value for the attribute ***post*** as shown below. Refresh the browser and confirm the ***PostSummaryItem*** displays as expected.

```
import PostSummaryItem from "./post-summary-list/post-summary-item";
const Tuiter = () => {
  return(
    <>
      <PostSummaryItem post={{
        "topic": "Web Development",
        "userName": "",
        "title": "jQuery",
        "time": "last week",
        "image": "../../../images/jquery.png",
        "tweets": "122K"
      }}/>
      <NavigationSidebar active="home"/>
      ...
    </>
  )
};
export default Tuiter;
```

PostSummaryList

The ***PostSummaryList*** is very similar to the ***WhoToFollowList***. Convert ***PostSummaryList*** into a React.js component by applying the same strategies applied to convert ***WhoToFollowList*** into a React.js component. In directory ***src/components/tuiter/post-summary-list***, copy the ***index.js*** from the previous assignment and convert it into a React.js compoment. You'll need to convert the ***posts.js*** file into a JSON file ***posts.json***. Once you're done, import the ***PostSummaryList*** component into the ***tuiter.js*** file, refresh and confirm the component renders as expected.

ExploreComponent

In directory ***src/components/tuiter/explore-screen***, copy the ***explore-component.js*** from the previous assignment and use the strategies you've practiced so far to convert it into a React.js compoment. Here's a list of things to remember when porting JavaScript components to React.js equivalent components.

- Import React.js
- Remove JavaScript extensions since they are optional
- Object deconstructed parameter
- Provide default values for object parameters
- Remove extra quotes (`)
- Remove unnecessary $ symbols
- Wrap in fragment tag
- Replace class with className
- If attribute value is dynamic, use expressions
- When loading data, convert it into properly formatted JSON files

- When mapping over arrays, replace function calls with tags
- Pass parameters as attribute values
- Convert all component function calls into equivalent tags
- Test the component by rendering in a parent component, e.g., ***tuiter.js***

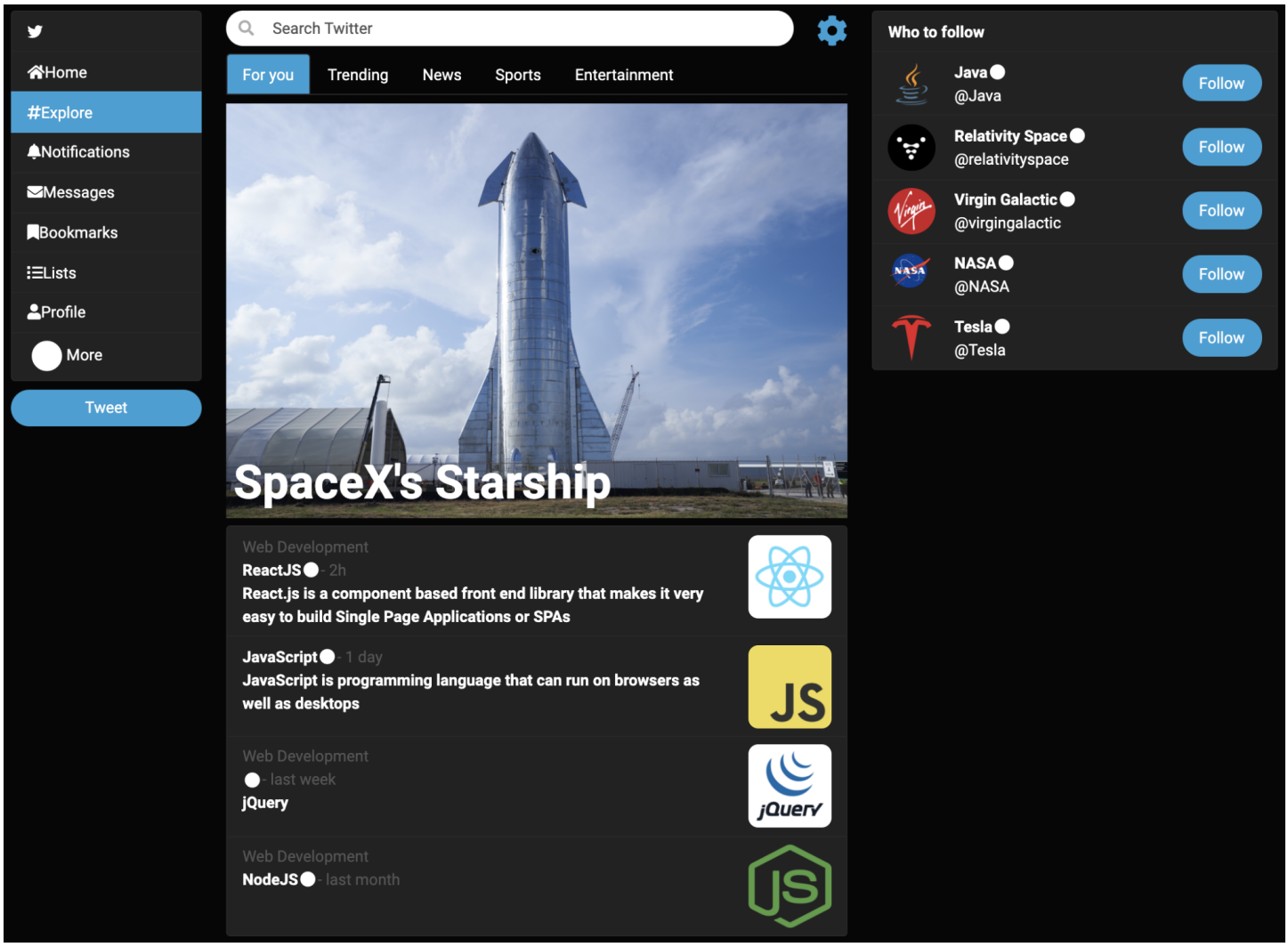
Remember that the ***ExploreComponent*** uses the ***PostSummaryList*** component towards the end.

ExploreScreen

The ***ExploreScreen*** consists of a parent component that brings all the other components into a single screen. It imports the ***NavigationSidebar*** and renders it on the left hand side. The ***WhoToFollowList*** is imported and rendered on the right hand side. The ***ExploreComponent*** is imported and rendered in the center of the screen. Below is an example of what the component might look like.

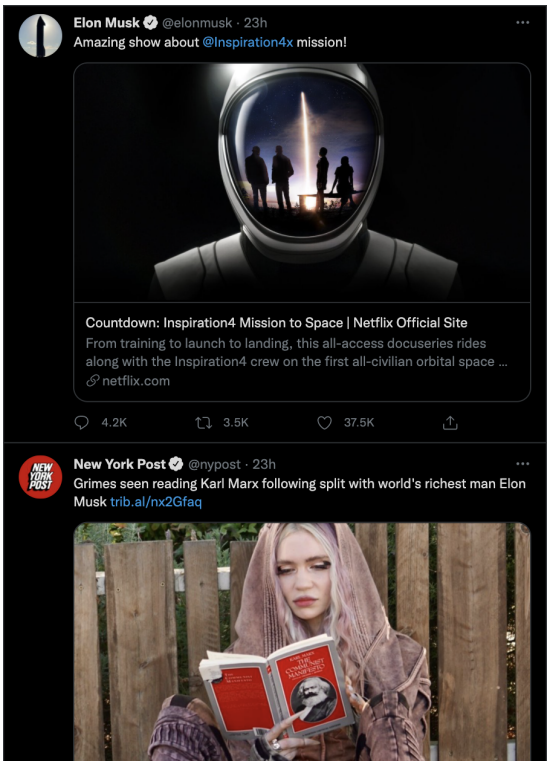
```
import React from "react";
import NavigationSidebar from "../navigation-sidebar";
import ExploreComponent from "./explore-component";
import WhoToFollowList from "../who-to-follow-list";
const ExploreScreen = () => {
   return(
      <div className="row mt-2">
         <div className="col-2 col-md-2 col-lg-1 col-xl-2">
            <NavigationSidebar active="explore"/>
         </div>
         <div className="col-10 col-md-10 col-lg-7 col-xl-6"
              style={{"position": "relative"}}>
            <ExploreComponent/>
         </div>
         <div className="d-sm-none d-md-none d-lg-block col-lg-4 col-xl-4">
            <WhoToFollowList/>
         </div>
      </div>
   );
};
export default ExploreScreen;
```

Import the ***ExploreScreen*** component into the ***tuiter.js*** component and confirm it renders as shown below. Comment out or remove all other components you had imported earlier since it was just for testing.



Challenge (required for graduate students)

Port the HomeScreen



In a new folder called ***/scr/components/tuiter/home-screen***, port the JavaScript ***HomeScreen*** component implemented in an earlier assignment and rendered here on the right.

Refactor navigation

The ***tuiter.js*** component we've been using to create the Tuiter clone allowed us to incrementally experiment with all the subcomponents. Now that we've got the hang of it, we can work with the components themselves directly. We're going to abandon the ***tuiter.js*** file and reimplement navigation with the ***HomeScreen*** and ***ExploreScreen*** components directly. In ***App.js***, create routes for the ***HomeScreen*** and ***ExploreScreen*** so that they are mapped to ***/tuiter/home*** and ***/tuiter/explore*** respectively. Restart the

application and confirm that navigating to http://localhost:3000/tuiter/home displays the ***HomeScreen*** and navigating to http://localhost:3000/tuiter/explore displays the ***ExploreScreen***. Use the code below for guidance.

```
<Route path="/tuiter/home" element={<HomeScreen/>}/>
<Route path="/tuiter/explore" element={<ExploreScreen/>}/>
```

Now, refactor ***NavigationSideBar*** hyperlinks to use ***Link*** elements instead. Use the ***to*** attributes to point to the ***HomeScreen*** and ***ExploreScreen*** paths as shown below. Point the first link to the root ('/') or ***/labs*** path so you can navigate to the Labs component. Use the code below for guidance.

```
import {Link} from "react-router-dom";
const NavigationSidebar = ( … ) => {
   return(
      <>
         <div className="list-group">
            <Link to="/"
                  className="list-group-item">
               <i className="fab fa-twitter"></i>
            </Link>
            <Link to="/tuiter/home"
                  className={`list-group-item ${active === 'home' ? 'active' : ''}`}>
               <i className="fa fa-home"></i>
               <span className="d-none d-xl-inline">Home</span>
            </Link>
            <Link to="/tuiter/explore"
                  className={`list-group-item ${active === 'explore' ? 'active' : ''}`}>
               <i className="fa fa-hashtag"></i>
               <span className="d-none d-xl-inline">Explore</span>
            </Link>
            …
      </>
   );
}
```

Make sure to update links in ***labs/index.js*** and ***hello-world.js*** so ***Tuiter*** link refers to ***/tuiter/home***.

Deliverables

As a deliverable, make sure you complete the ***Labs***, ***Tuiter*** and ***Challange*** (if graduate student) sections of this assignment. All your work must be done in a branch called ***a6***. When done, add, commit and push the branch to GitHub. Deploy the new branch to Netlify and confirm it's available in a new URL based on the branch name. Submit the link to your GitHub repository and the new URL where the branch deployed to in Netlify. Here's an example on the steps:

*Create a branch called **a6***

```
git  checkout  -b  a6
```

```
# do all your work
```

Do all your work, e.g., ***Labs*** exercises, ***Tuiter***, ***Challenge*** (graduate students)

Add, commit and push the new branch

```
git  add  .
git  commit  -am  "a6 React.js sp22"
git  push
```

If you have ***Netlify*** configured to auto deploy, then confirm it auto deployed. If not, then deploy the branch manually.

In Canvas, submit the following

1. The new URL where your ***a6*** branch deployed to on Netlify
2. The link to your new branch in GitHub.