Creating Single Page Applications with React.js

1 Introduction

In this assignment we introduce **JavaScript** as the language for programming the browser to implement **React.js Single Page Applications** (**SPAs**).

2 Labs

This section presents **JavaScript** and **React.js** examples to program the browser, interact with the user, and generate dynamic user interfaces. Use the same project you worked on in the last assignment. After you work through the examples you will apply the skills while creating a clone of **Canvas** on your own. Using **IntelliJ** or **VS Code** open the project you created in the previous assignment, **kanbas-react-web-app**. Do all your work under the **src** directory of your project.

2.1 Implementing Single Page Applications

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 is rendered by the **ReactDOM** library into a DIV with a **root** ID in the **public/index.html** document. Make sure **public/index.html** contains the **div#root** as shown below. Remove or comment all other content in the **body** tag from previous assignments.

```
public/index.html
```
```html
<html>
  <body>
    <!-- uncomment the following div and comment out all other content within the body -->
    <div id="root"></div>
    <!-- Remove or comment everything else from previous assignments -->
  </body>
</html>
```
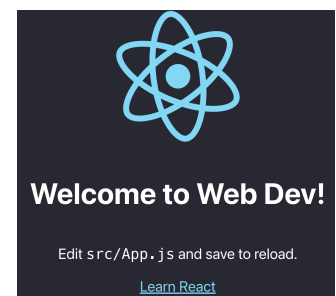
The **React.js** application is implemented in **src/index.tsx** importing **React** and **ReactDOM** libraries as shown below. If you had commented anything in **src/index.tsx**, uncomment it so that when you run the project it renders as a default React application. **ReactDOM** uses **document.getElementById('root')** to retrieve a reference to the **DOM** element declared in **index.html**. **ReactDOM** then creates an instance of the **App** component and appends its output to the element whose ID is **root**. The **src/App.tsx** is the entry point of the **React.js** application we're going to build and it contains code generated by the **create-react-app** tool we used to create the project at the beginning of the course.

```
src/index.tsx                                          Browser
```
```tsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App'; // imports from App.tsx. The .tsx extension is implied
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
 <React.StrictMode>
   <App />
 </React.StrictMode>
);
```

Welcome to Web Dev!

Edit src/App.js and save to reload.

Learn React

The *import* statement that imports *App* in *index.tsx*, imports from *'./App'* which references file *App.tsx*. The *.tsx* extension is optional in import statements. The content of *index.tsx*, and other *.tsx* files, is a mixture of *TypeScript* (*ts*) and *XML* (*x*). *TypeScript* is a superset of *JavaScript*, extending the language to support type safety. We'll often use *TypeScript* and *JavaScript* interchangeably. Let's replace the content of *src/App.tsx* with the code below. It's basically a function called *App* that returns an *H1* heading element greeting the world. Note how the *return* statement is returning an *HTML tag*, seamlessly mixing *TypeScript* and *HTML*. This is possible because *React.js* uses a library called *JSX* or *JavaScript XML* allowing 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.

src/App.tsx	*Browser*

```
import './index.css';
function App() {
    return (
        <h1>Hello World!</h1>
    );
}
export default App;
```

Hello World!

To test, start the *React* application using *npm* as shown below. Run the command from the root directory of your project. Confirm the browser refreshes with the *Hello World!* message.

```
npm  start
```

2.1.1 Installing CSS libraries Bootstrap and React Icons

If you have not yet installed Bootstrap, install it from the root of your project as follows.

```
npm  install  bootstrap
```

Let's also install the React icons library. This is an alternative to the Fontawesome icon library used in previous assignments.

```
npm  install  react-icons
```

Once the libraries are installed you can load them by importing them from the *src/index.tsx* as shown below. We'll use the icons in later exercises. Confirm that the browser refreshes with *bootstrap* styling.

src/index.tsx

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import 'bootstrap/dist/css/bootstrap.min.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
 <React.StrictMode>
   <App />
 </React.StrictMode>
);
```

2.1.2 Implementing the Labs component

Let's create a **Labs** component to work on all the lab exercises. Create a folder called **src/Labs** and then a **TypeScript** file in **src/Labs/index.tsx**. Add the following content in the new **index.tsx** file and import the new component in the **App.tsx**.

---

src/Labs/index.tsx

```
function Labs() {
    return (
        <div>
            <h1>Assignment 3</h1>
        </div>
    );
}
export default Labs;
```

In **App.tsx**, import the **Labs** component as shown below. Wrap the HTML content in a DIV element. Confirm the application renders as shown below. Notice that importing **Labs/index.tsx** only requires importing the **Labs** directory.

---

src/App.tsx   **Browser**

```
import Labs from "./Labs";        // imports ./Labs/index.tsx
function App() {
    return (                      // return can only return a single component
        <div>                     // that's why we wrap the whole content with
            <Labs/>               // a <div>
            <h1>Hello World!</h1>
        </div>
    );}
```

Assignment 3
Hello World!

2.1.3 Breaking out assignments into separate components

The **Labs** component will hold all the lab exercises for this assignment as well as future assignments. Let's break out each assignment into its own separate component. In a new file in **src/Labs/a3/index.tsx**, create the following component. Note that in React.js we use **className** instead of **class** since **class** is a **JavaScript** reserved keyword.

---

src/Labs/a3/index.tsx

```
function Assignment3() {
    return (
        <div className="container">      // "container" is a Bootstrap class. The library was imported in src/index.tsx
            <h1>Assignment 3</h1>        // and the classes are available to all components without needing to import
        </div>                          // it again. You can also create additional CSS files and import them in
    );                                  // TSX files that need specific styling.
}
export default Assignment3;
```

Then import the new component into the **Labs** component. Confirm the application renders as before. In later assignments you'll be creating separate components, one for each assignment, that contain the exercises for that specific assignment. You'll import them into the **Labs** component so they are all accessible in one place.

---

src/Labs/index.tsx   **Browser**

```
import Assignment3 from "./a3";
function Labs() {
    return (
        <div>
            <h1>Assignment 3</h1>
            <Assignment3/>
        </div>
    );}
```

Assignment 3
Hello World!

2.1.4 Breaking out Hello World into a separate component

React.js incourages breaking up large applications into smaller parts or *components* you can then assemble into complex user interfaces. Let's create another *React.js component* by breaking out the *Hello World* H1 element into a separate *TypeScript* file as shown below. In *src/Labs/a3/HelloWorld.tsx* create a *HelloWorld* component as shown below.

```
src/Labs/a3/HelloWorld.tsx

function HelloWorld() {           // notice that if we're returning a single line of code, the parenthesis
    return <h1>Hello World!</h1>; // in the return statement are optional. If HelloWorld would have been a
};                                // more complex component, we could have implemented it in
export default HelloWorld;        // HelloWorld/index.tsx giving us a whole directory to implement it
```

We can then import the new component in *src/App.tsx* as shown below. Note the missing *.tsx* 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.

```
src/App.tsx

import Labs from "./Labs";
import HelloWorld from "./Labs/a3/HelloWorld";   // if the HelloWorld component would have been implemented in
function App() {                                 // HelloWorld/index.tsx, the import statement would have been the
    return (                                     // same. This simplifies deciding implementing components as single
        <div>                                    // file or a folder. If the extension is ommitted from the import
            <Labs/>                              // statement, then first it will attempt to import a file called
            <HelloWorld/>                        // HelloWorld.tsx. If it fails it will then attempt to import
        </div>                                   // index.tsx in a folder called HelloWorld, e.g., HelloWorld/tsx.
    );                                           // Since the HelloWorld component is trivial, we decided to use
}                                                // a single file, e.g., HelloWorld.tsx
```

2.1.5 Creating a Kanbas placeholder component

Let's create a nother component we'll use later to implement the *Kanbas* application. Let's create the component in *src/Kanbas/index.tsx* with the content below. This will be a placeholder for a later section.

```
src/Kanbas/index.tsx

function Kanbas() {          // since the Kanbas component consists of an entire application with lots of screens
    return(                  // each implemented in several files, we've decided to use an entire folder to implement
        <div>                // the component. It is common use the same name for the folder and component name, but
            <h1>Kanbas</h1>  // it is not required.
        </div>
    );
}
export default Kanbas
```

Import the new Kanbas component in *App.tsx* as shown below. Confirm the output renders as shown below.

| src/App.tsx		Browser

```import Kanbas from "./Kanbas";function App() { return (<Labs/> <Kanbas/> <HelloWorld/>);}```	`// this import statement will first attempt to import`
	`// a file called Kanbas.tsx, and if it fails it will`
	`// then attempt to import index.tsx in a folder called`
	`// Kanbas, e.g., Kanbas/index.tsx, which is what we`
	`// have here`
		Assignment 3
		Kanbas
		Hello World!

2.2 Learning about JavaScript

In the following exercises, we'll learn about the **JavaScript** programming language. We'll create a **JavaScript** component where we can copy and paste the following exercises as we learn about the different features of the language. Create **src/Labs/a3/JavaScript/index.tsx** and import it from the **Assignment3** component as shown below
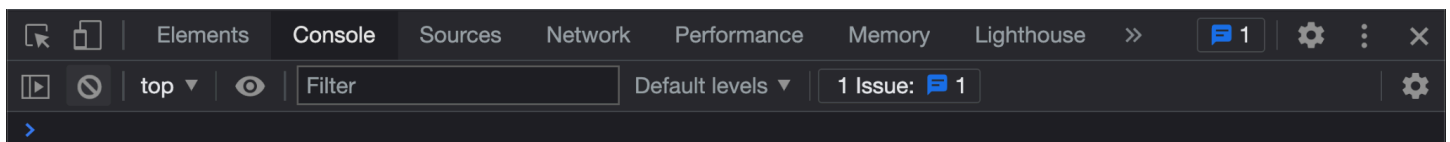
src/Labs/a3/JavaScript/index.tsx	src/Labs/a3/index.tsx

``
function JavaScript() {
    return(
 	
             JavaScript
 	
    );
}
export default JavaScript
``
	``
	import JavaScript from "./JavaScript";
	function Assignment3() {
	 return (
	   <div className="container">
	     <h1>Assignment 3</h1>
	     <JavaScript/>
	   </div>
	);
	}
	``

2.2.1 Writing to the Console from JavaScript

A useful feature of modern browsers is providing a development environment where developers can analyze the performance of their scripts. One way to analyze scripts are behaving correctly is to write output to the **console** from within scripts. To practice writing to the **console**, bring up the console on the browser by right clicking on the page and selecting **Inspect**. The page will split in half displaying useful developer tools similar as shown below.
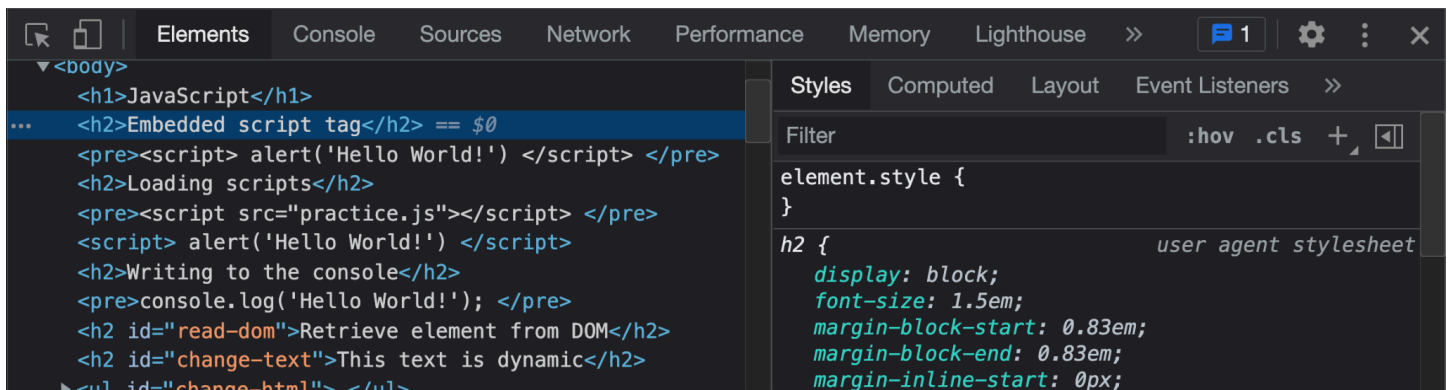
Click on the **Console** tab where we will be logging to throughout this assignment.

Add a **console.log()** statement to the **JavaScript** component, reload the screen and confirm that "**Hello World!**" is displayed in the console.

src/Labs/a3/JavaScript/index.tsx	Console

``
function JavaScript() {
    console.log('Hello World!');
    return(
 	
             JavaScript
 	
    );
}
``
	Hello World!

2.2.2 Variables and Constants

Variables can store state information about applications. To practice declaring variables and constants, create the **VariablesAndConstants** component below and import it from the **JavaScript** component. Confirm the browser displays as shown on the right. We'll be creating several components to practice various features of the **JavaScript** language. Import them into the **JavaScript** component and confirm the output is as described for each of the lab exercises.

JavaScript

Variables and Constants

functionScoped = 2

blockScoped = 5

constant1 = -3

src/Labs/a3/JavaScript/variables/VariablesAndConstants.tsx

```
function VariablesAndConstants() {
   var functionScoped = 2;
   let blockScoped = 5;
   const constant1 = functionScoped - blockScoped;

   return(
      <div>
         <h2>Variables and Constants</h2>
         functionScoped = { functionScoped }<br/>
         blockScoped = { blockScoped }<br/>
         constant1 = { constant1 }<br/>
      </div>
   );
}
export default VariablesAndConstants
```

src/Labs/a3/JavaScript/index.tsx

```
import VariablesAndConstants
   from "./variables/VariablesAndConstants";

function JavaScript() {
   console.log('Hello World!');
   return(
      <div>
         <h1>JavaScript</h1>
         <VariablesAndConstants/>
      </div>
   );
}
export default JavaScript
```

2.2.3 Variable Types

JavaScript declares several datatypes such as **Number**, **String**, **Date**, and so on. To practice with variable types, create the **VariableTypes** component shown below and import it from the **JavaScript** component. Confirm that the browser renders as shown below on the right. Note that we had to convert the boolean variable into a string type before it could render in the browser.

src/Labs/a3/JavaScript/variables/VariableTypes.tsx

```
function VariableTypes() {
   let numberVariable = 123;
   let floatingPointNumber = 234.345;
   let stringVariable = 'Hello World!';
   let booleanVariable = true;
   let isNumber = typeof numberVariable;
   let isString = typeof stringVariable;
   let isBoolean = typeof booleanVariable;
   return(
      <div>
         <h2>Variables Types</h2>
         numberVariable = { numberVariable }<br/>
         floatingPointNumber = { floatingPointNumber }<br/>
         stringVariable = { stringVariable }<br/>
         booleanVariable = { booleanVariable + "" }<br/>
         isNumber = { isNumber }<br/>
         isString = { isString }<br/>
         isBoolean = { isBoolean }<br/>
      </div>
   );
}
export default VariableTypes
```

Browser

Variables Types

numberVariable = 123

floatingPointNumber = 234.345

stringVariable = Hello World!

booleanVariable = true

isNumber = number

isString = string

isBoolean = boolean

2.2.4 Boolean Variables

To practice with Boolean data types, create a component called **BooleanVariables** and import it in the **JavaScript**. Use the previous lab exercises as a guide of how to complete this exercise. The new component should add a new section called **Boolean Variables** that displays each of the new variables so that the browser renders as shown below on the right. You might need to cast the boolean values to string by concatenating an empty string to display the variables, e.g., **false3 = {false3 + ""}
. Add **function, **export**, and other keywords as needed.

src/Labs/a3/JavaScript/variables/BooleanVariables.tsx	Browser
| ```tsx
let numberVariable = 123, floatingPointNumber = 234.345;
let true1 = true, false1 = false;
let false2 = true1 && false1;
let true2 = true1 || false1;
let true3 = !false2;
let true4 = numberVariable === 123; // always use === not ==
let true5 = floatingPointNumber !== 321.432;
let false3 = numberVariable < 100;
return (
  <div>
    <h2>Boolean Variables</h2>
    true1     = {true1 + ""}      <br />
    false1    = {false1 + ""}     <br />
    false2    = {false2 + ""}     <br />
    true2     = {true2 + ""}      <br />
    true3     = {true3 + ""}      <br />
    true4     = {true4 + ""}      <br />
    true5     = {true5 + ""}      <br />
    false3    = {false3 + ""}     <br />
  </div>
);
``` | **Boolean Variables**<br><br>true1 = true<br>false1 = false<br>false2 = false<br>true2 = true<br>true3 = true<br>true4 = true<br>true5 = true<br>false3 = false |

2.2.5 Conditionals

Conditionals allow scripts to make decisions based on some predicate that compares values and variables. Scripts can decide to execute different parts of the code based on the result of these predicates using the **if/else** and other constructs. Create the following components and import them into the **JavaScript** component. Confirm that the components render as shown.

2.2.5.1 If Else

The most common use of conditionals is **if/else** statements that evaluate a predicate and can decide to execute one of two different code blocks depending on whether the predicate evaluates to **true** or **false**. To practice with **if/else**, create a component called **IfElse** based on the code shown below. It should render a new section labeled **If Else** and render as shown below. The **true1** paragraph is only rendered if **true1** is true. The **ternary operators** ? and : can be used to render one of two options based on the value of a boolean expression.

src/Labs/a3/JavaScript/conditionals/IfElse.tsx	Browser
| ```tsx
let true1 = true, false1 = false;
...
return(
  <div>
    <h2>If Else</h2>
    { true1 && <p>true1</p> }
    { !false1 ? <p>!false1</p> : <p>false1</p> }
  </div>
)
``` | **If Else**<br><br>true1<br><br>!false1 |

2.2.5.2 Ternary Conditional Operator

Ternary conditional operators are concise alternative to if statements. It takes three arguments
1. A conditional expression that evaluates to true or false followed by a question mark (?)
2. An expression that evaluates if the conditional is true followed by a colon (:)
3. Followed by an expression that evaluates iff the conditional is false

To practice the ternary operator, create a new component called **TernaryOperator** based on the code shown below which should render as shown below on the right.

src/Labs/a3/JavaScript/conditionals/TernaryOperator.tsx	Browser
```let loggedIn = true;\n...\nreturn(\n \n Logged In \n { loggedIn ? Welcome : Please login }\n \n)```	# Logged In\n\nWelcome

2.2.6 Working with functions

Functions allow reusing an algorithm by wrapping it in a named, parameterized block of code. JavaScript supports two styles of functions based on the history of language. Create a component called **WorkingWithFunctions** and import it to the **JavaScript** component. Implement the following exercises in this new component.

2.2.6.1 Legacy ES5 function

Declaring functions consists of wrapping a block of code, naming it, and declaring parameters as shown below. In ECMAScript 5 (ES5) and earlier, the syntax for functions is

```
function <functionName> (<parameterList>) { <functionBody> }
```

To practice using functions create a new component called **ES5Functions** based on the code below. Import this new component in **WorkingWithFunctions** and confirm the browser renders as shown.

src/Labs/a3/JavaScript/functions/ES5Functions.tsx	Browser
```function add (a: number, b: number) {\n return a + b;\n}\nconst twoPlusFour = add(2, 4);\nconsole.log(twoPlusFour);\n\nreturn (\n <>\n Functions \n Legacy ES5 functions \n twoPlusFour = { twoPlusFour } \n add(2, 4) = { add(2, 4) } \n </>\n)```	# Functions\n\n## Legacy ES5 functions\n\ntwoPlusFour = 6\nadd(2, 4) = 6

2.2.6.2 ES6 arrow functions

A new version of **JavaScript** was introduced in 2015 and is officially referred to as **ECMAScript 6** or **ES6**. A new syntax for declaring functions was introduced which is less verbose and provides tons of new features we'll explore throughout this course. This function syntax is often referred to as "arrow functions". To practice using ES6 functions, create a new

component called **ArrowFunctions** based on the code below. Import this new component in **WorkingWithFunctions** and confirm the browser renders as shown.

src/Labs/a3/JavaScript/functions/ArrowFunctions.tsx	Browser
```const subtract = (a: number, b: number) => { return a - b;}const threeMinusOne = subtract(3, 1);console.log(threeMinusOne);return (<> New ES6 arrow functions threeMinusOne = {threeMinusOne} subtract(3, 1) = {subtract(3, 1)} </>)```	## New ES6 arrow functions threeMinusOne = 2 subtract(3, 1) = 2

NOTE: Throughout the last couple of exercises we've provided code in the return statement to render the variables in the browser and asked that you confirm the output matches. Going forward we'll omit the return statement, but please continue to implement it and confirm the output matches the one provided.

2.2.6.3 Implied returns

One of the new features of the new ES6 functions is **implied returns**, that is, if the body of the function consists of just returning some value or expression, then the return statement is optional and can be replaced with just the value or expression. To practice this feature create a new component called **ImpliedReturn** based on the code below. Import this new component in **WorkingWithFunctions** and confirm the browser renders as shown.

src/Labs/a3/JavaScript/functions/ImpliedReturn.tsx	Browser
```const multiply = (a: number, b: number) => a * b;const fourTimesFive = multiply(4, 5);console.log(fourTimesFive);```	## Implied return fourTimesFive = 20 multiply(4, 5) = 20

2.2.6.4 Optional parenthesis and parameters

Another new feature is optional parameter parenthesis if functions have only one parameter. To practice this new feature create a new component called **FunctionParenthesisAndParameters** based on the code below. Import this new component in **WorkingWithFunctions** and confirm the browser renders as shown.

src/Labs/a3/JavaScript/functions/FunctionParenthesisAndParameters.tsx	Browser
```const square = (a: number) => a * a;const plusOne = (a: number) => a + 1;const twoSquared = square(2);const threePlusOne = plusOne(3);```	## Parenthesis and parameters twoSquared = 4 square(2) = 4 threePlusOne = 4 plusOne(3) = 4

2.2.7 Working with Arrays

Arrays can group together several values into a single variable. Arrays can group together values of different datatypes, e.g., number arrays, string arrays, and even a mix and match of datatypes in the same array. Not that you would ever want to do that. To practice with arrays, copy and paste the code below to the end of **index.tsx**. Use **console.log()** to print the

title of this section and then all the variables and constants shown below. Confirm the console displays as shown on the right below. The numbers in parenthesis at the beginning of a line is the length of the array. The numbers and colons at the beginning of a line are the indices of the element. You can ignore these. Create a component called **WorkingWithArrays** and import it to the **JavaScript** component. Use the code below as a guide to rendering the content on the right.

src/Labs/a3/JavaScript/arrays/WorkingWithArrays.tsx	Browser
| ```javascript
var functionScoped = 2;
let blockScoped = 5;
const constant1 = functionScoped - blockScoped;
let numberArray1 = [1, 2, 3, 4, 5];
let stringArray1 = ['string1', 'string2'];

let variableArray1 = [
    functionScoped,   blockScoped,
    constant1,        numberArray1,   stringArray1
];
``` | **Working with Arrays**<br><br>numberArray1 = 12345<br><br>stringArray1 = string1string2<br><br>variableArray1 = 25-312345string1string2 |

Implement the following exercises and import them to this new component confirming they render as shown.

2.2.7.1 Array index and length

The length of an array is available as property **length** in the array variable. The **indexOf()** function allows finding where a particular array member is found. To practice with array indices and length, implement a new component called **ArrayIndexAndLength** based on the code below. Import this new component in **WorkingWithArrays** and confirm the browser renders as shown.

src/Labs/a3/JavaScript/arrays/ArrayIndexAndLength.tsx	Browser
| ```javascript
let numberArray1 = [1, 2, 3, 4, 5];
const length1 = numberArray1.length;
const index1 = numberArray1.indexOf(3);
``` | **Array index and length**<br><br>length1 = 5<br><br>index1 = 2 |

2.2.7.2 Adding and Removing Data to/from Arrays

In most languages arrays are immutable, whereas in JavaScript we can easily add or remove elements from the array. The **push()** function appends an element at the end of an array. The **splice()** function can remove/add an element anywhere in the array. To practice adding and removing data from arrays, implement a new component called **AddingAndRemovingDataToFromArrays** based on the code below. Import this new component in **WorkingWithArrays** and confirm the browser renders as shown.

src/Labs/a3/JavaScript/arrays/AddingAndRemovingDataToFromArrays.tsx	Browser
| ```javascript
let numberArray1 = [1, 2, 3, 4, 5];
let stringArray1 = ['string1', 'string2'];
numberArray1.push(6);          // adding new items
stringArray1.push('string3');
numberArray1.splice(2, 1);     // remove 1 item starting at 2
stringArray1.splice(1, 1);
``` | **Add and remove data to arrays**<br><br>numberArray1 = 12456<br><br>stringArray1 = string1string3 |

2.2.7.3 For Loops

We can operate on each array value by iterating over them in a **for loop**. To practice with for loops, implement a new component called **ForLoops** based on the code below. Import this new component in **WorkingWithArrays** and confirm the browser renders as shown.

src/Labs/a3/JavaScript/arrays/ForLoops.tsx	Browser

``tsx
let stringArray1 = ['string1', 'string3'];
let stringArray2 = [];
for (let i = 0;
     i < stringArray1.length;
     i++) {
  const string1 = stringArray1[i];
  stringArray2.push(
    string1.toUpperCase());
}
``
	# Looping through arrays
	
	stringArray2 = STRING1STRING3

2.2.7.4 The Map Function

An array's **map** function can iterate over an array's values, apply a function to each value, and collate all the results in a new array. The first example below iterates over the **numberArray1** and calls the **square** function for each element. The **square** function was declared earlier in this document and it accepts a parameter and returns the square of the parameter. The **map** function collates all the squares into a new array called **squares** as shown below. The second example does the same thing, but uses a function that calculates the **cubes** of all numbers in the same **numberArray1** array. To practice with **map**, implement a new component called **MapFunction** based on the code below. Import this new component in **WorkingWithArrays** and confirm the browser renders as shown.

src/Labs/a3/JavaScript/arrays/MapFunction.tsx	Browser

``tsx
let numberArray1 = [1, 2, 3, 4, 5, 6];
const square = (a: number) => a * a;
 	
const squares = numberArray1.map(square);
const cubes = numberArray1.map(a => a * a * a);
``
	## Map
	
	squares = 14162536
	
	cubes = 1864125216

2.2.7.5 JSON Stringify

JavaScript has a global object called **JSON** which stands for **JavaScript Object Notation**. The object provides several useful formatting functions such as **stringify()** and **parse()**. Stringify converts JavaScript variables to formatted strings. For instance let's format the following arrays and display them in the browser. Note how the array is rendered with square brackets and items are separated by commas.

src/Labs/a3/JavaScript/json/JsonStringify.tsx	Browser

``tsx
function JsonStringify() {
  const squares = [1, 4, 16, 25, 36];
  return (
    <>
       JSON Stringify
      squares = {JSON.stringify(squares)}
    </>
  );
}
export default JsonStringify;
``
	# JSON Stringify
	
	squares = [1,4,16,25,36]

2.2.7.6 The Find Function

An array's **find** function can search for an item in an array and return the element it finds. The find function takes a function as an argument that serves as a predicate. The predicate should return true if the element is the one you're looking for. The predicate function is invoked for each of the elements in the array and when the function returns true, the find function stops because it has found the element that it was looking for. To practice, implement a new component called **FindFunction** based on the code below. Import this new component in **WorkingWithArrays** and confirm the browser renders as shown.

src/Labs/a3/JavaScript/arrays/FindFunction.tsx	Browser
| ```
let numberArray1 = [1, 2, 3, 4, 5];
let stringArray1 = ['string1', 'string2', 'string3'];

const four = numberArray1.find(a => a === 4);
const string3 = stringArray1.find(a => a === 'string3');
``` | **Find function**<br><br>four = 4<br>string3 = string3 |

2.2.7.7 The Find Index Function

Alternatively we can use **findIndex** function to determine the index where an element is located inside an array. Copy the code and display the content as shown below.

src/Labs/a3/JavaScript/arrays/FindIndex.tsx	Browser
| ```
let numberArray1 = [1, 2, 4, 5, 6];
let stringArray1 = ['string1', 'string3'];

const fourIndex = numberArray1.findIndex(a => a === 4);
const string3Index = stringArray1.findIndex(a => a === 'string3');
``` | **FindIndex function**<br><br>fourIndex = 2<br>string3Index = 1 |

2.2.7.8 The Filter Function

The **filter** function can look for elements that meet a criteria and collate them into a new array. For instance, the example below is looking through the **numberArray1** array for all values that are greater than 2. Then we look for all even numbers and then for all odd numbers. All the results are stored in corresponding arrays with appropriate names. To practice, implement a new component called **FilterFunction** based on the code below. Import this new component in **WorkingWithArrays** and confirm the browser renders as shown.

src/Labs/a3/JavaScript/arrays/FilterFunction.tsx	Browser
| ```
let numberArray1 = [1, 2, 4, 5, 6];
const numbersGreaterThan2 = numberArray1.filter(a => a > 2);
const evenNumbers = numberArray1.filter(a => a % 2 === 0);
const oddNumbers = numberArray1.filter(a => a % 2 !== 0);
``` | Filter function<br><br>numbersGreaterThan2 = 456<br>evenNumbers = 246<br>oddNumbers = 15 |

2.2.8 Template Literals

Generating dynamic HTML consists of writing code that manipulates and concatenates strings to generate new HTML strings based on some program logic. Basically consists of one language writing code in another language, much what a compiler does. Working with strings can be error prone especially if you have to use lots of extra operations and variables to concatenate the resulting string. JavaScript template strings provide a better approach by allowing embedding expressions and algorithms right within strings themselves. To practice, implement a new component called **TemplateLiterals** based on the code below. Import this new component in **JavaScript** and confirm the browser renders as shown.

src/Labs/a3/JavaScript/string/TemplateLiterals.tsx	Browser
| ```
const five = 2 + 3;
const result1 = "2 + 3 = " + five;
const result2 = `2 + 3 = ${2 + 3}`;
const username = 'alice';
const greeting1 = `Welcome home ${username}`;
const loggedIn = false;
const greeting2 = `Logged in: ${loggedIn ? "Yes" : "No"}`;
``` | **Template Literals**<br><br>result1 = 2 + 3 = 5<br>result2 = 2 + 3 = 5<br>greeting1 = Welcome home alice<br>greeting2 = Logged in: No |

2.2.9 JavaScript Object Notation (JSON)

Multiple values, of various datatypes can be combined together to create complex datatypes called **objects**. For example the code below declares a **house** object collecting several numbers, strings, arrays, and other objects to represent a particular instance of a house. The **house** variable is assigned an **object literal** declared within opening and closing curly braces **{** and **}**. Objects contain pairs of **properties** and values separated by commas. Values can be of any datatype including **Number**, **String**, **Boolean**, arrays and other objects. In the example below we declared a **house** with 4 **bedrooms**, 2.5 **bathrooms** and 2000 **squareFeet**. The house has a nested object stored in property **address** which contains **String** properties such as **street**, **city** and **state**. The **owners String** array declares the names of the owners. To practice with JSON, create a **House** component as shown below, import it into the **JavaScript** component, and confirm it renders as shown below.

src/Labs/a3/JavaScript/json/House.tsx	Browser

<pre>function House() {
  const house = {
    bedrooms: 4,
    bathrooms: 2.5,
    squareFeet: 2000,
    address: {
      street: "Via Roma",
      city: "Roma",
      state: "RM",
      zip: "00100",
      country: "Italy",
    },
    owners: ["Alice", "Bob"],
  };
  return (
    <div>
      <h2>House</h2>
      <h3>bedrooms</h3>
      {house.bedrooms}
      <h3>bathrooms</h3>
      {house.bathrooms}
      <h3>Data</h3>
      <pre>{JSON.stringify(house, null, 2)}</pre>
    </div>
  );
}
export default House;</pre>
	<pre># House
	## bedrooms
	4
	## bathrooms
	2.5
	## Data
	{
	  "bedrooms": 4,
	  "bathrooms": 2.5,
	  "squareFeet": 2000,
	  "address": {
	    "street": "Via Roma",
	    "city": "Roma",
	    "state": "RM",
	    "zip": "00100",
	    "country": "Italy"
	  },
	  "owners": [
	    "Alice",
	    "Bob"
	  ]
	}</pre>

2.2.10 The Spread Operator

The spread operator (...) is used to expand an iterable object or array into another object or array. In the example below we declare array **arr1** and then expand its content (spread) into array **arr2**. The resulting array **arr2** contains the contents of **arr1**, followed by the rest of the items declared in **arr2**. The spread operator can also be applied to objects as illustrated in the following example. Below, **obj1** declares an object with three properties **a**, **b**, and **c**. We then spread **obj1** on **obj2** so that **obj2** ends up with the properties from both **obj1** and **obj2**. When declaring **obj3**, we first spread **obj1** and then declare **b** with a value of 4. Since **obj1** also has a property called **b** with a value of **2**, there is potential collision of properties in **obj3**. The collision is resolved by keeping the last declaration overriding any previous values, so **obj3.b** ends up being 4. To practice the spread operator, create the **Spread** component as shown below, import it in the **JavaScript** component and confirm it renders as shown below.

```tsx
function Spreading() {
 const arr1 = [ 1, 2, 3 ];
 const arr2 = [ ...arr1, 4, 5, 6 ];
 const obj1 = { a: 1, b: 2, c: 3 };
 const obj2 = { ...obj1, d: 4, e: 5, f: 6 };
 const obj3 = { ...obj1, b: 4 };
 return (
   <div>
     <h2>Spread Operator</h2>
     <h3>Array Spread</h3>
     arr1 = { JSON.stringify(arr1) }  <br />
     arr2 = { JSON.stringify(arr2) }  <br />
     <h3>Object Spread</h3>
     { JSON.stringify(obj1) }         <br />
     { JSON.stringify(obj2) }         <br />
     { JSON.stringify(obj3) }         <br />
   </div>);
}
export default Spreading;
```

Spread Operator

Array Spread

arr1 = [1,2,3]

arr2 = [1,2,3,4,5,6]

Object Spread

{"a":1,"b":2,"c":3}

{"a":1,"b":2,"c":3,"d":4,"e":5,"f":6}

{"a":1,"b":4,"c":3}

2.2.11 Destructing

While the spreader operator is used to expand an iterable object into the list of arguments, the destructing operator is used to unpack values from arrays, or properties from objects, into distinct variables. In the example below we declare object **person** and array **numbers**. These can be unpacked, or **destructed**, into new variables or constants by an object's property name or an array's item position. The curly brackets around constants **name** and **age**, destruct the object **person** on the right side of the assignment, and assigns the properties of the same name into the new constants. The constants **name** and **age** end up having the values of **person.name** and **person.age** respectively. Essentially it is the equivalent to

```
const name = person.name
const age = person.age
```

While object destructing is based on the names of the properties, destructing arrays is based on the positions of the items. In the example below, we declare the **numbers** array and then use the square brackets to destruct the array into new constants **first**, **second**, and **third**. These new constants end up with the values of **numbers[0]**, **numbers[1]**, and **numbers[2]**. Essentially it is equivalent to

```
const first = numbers[0]
const second = numbers[1]
const third = numbers[2]
```

To practice destructing objects and arrays, create component **Destructing** as shown below, import it in the **JavaScript** component, and confirm it renders as shown below.

src/Labs/a3/JavaScript/json/Destructing.tsx	Browser

```tsx
function Destructing() {
 const person = { name: "John", age: 25 };
 const { name, age } = person;
 const numbers = ["one", "two", "three"];
 const [ first, second, third ] = numbers;
 return (
   <div>
     <h2>Destructing</h2>
     <h3>Object Destructing</h3>
     const &#123; name, age &#125; =
         &#123; name: "John", age: 25 &#125;<br /><br />
     name = {name}<br />
     age = {age}
     <h3>Array Destructing</h3>
     const [first, second, third] = ["one","two","three"]<br/><br/>
     first = {first}<br />
     second = {second}<br />
     third = {third}
   </div>
 );
}
export default Destructing;
```

Destructing

Object Destructing

const { name, age } = { name: "John", age: 25 }

name = John

age = 25

Array Destructing

const [first, second, third] = ["one", "two", "three"]

first = one

second = two

third = three

2.2.12 Function Destructing

The destructing objects syntax is very popular in React.js, especially when passing parameters to functions. In the example below we declare two functions **add** and **subtract** using the new arrow function syntax. The **add** function takes two arguments **a** and **b** and returns the sum of the arguments. The **subtract** function takes a single object argument with properties **a** and **b** with values **4** and **2**. In the argument list declaration, **subtract** uses object destructing to declare constants **a** and **b** which unpacks the values **4** and **2** from the object argument with properties of the same name. To practice **function destructing**, copy the code below into a **FunctionDestructing** component, import it into the **JavaScript** component, and confirm it renders as shown below on the right.

src/Labs/a3/JavaScript/functions/FunctionDestructing.tsx	Browser

```tsx
function FunctionDestructing() {
 const add = (a: number, b: number) => a + b;
 const sum = add(1, 2);
 const subtract = ({ a, b }: { a: number; b: number }) => a - b;
 const difference = subtract({ a: 4, b: 2 });
 return (
   <div>
     <h2>Function Destructing</h2>
     const add = (a, b) =&gt; a + b;<br />
     const sum = add(1, 2);<br />
     const subtract = (&#123; a, b &#125;) =&gt; a - b;<br />
     const difference = subtract(&#123; a: 4, b: 2 &#125;);<br/>
     sum = {sum}<br />
     difference = {difference}
   </div>
);}
export default FunctionDestructing;
```

Function Destructing

const add = (a, b) => a + b;

const sum = add(1, 2);

const subtract = ({ a, b }) => a - b;

const difference = subtract({ a: 4, b: 2 });

sum = 3

difference = 2

2.3 Implementing Navigation in Single Page Applications

Earlier we mentioned that **Single Page Applications** (**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, let's implement navigation between the components we've created so far: **HelloWorld**, **Labs**, and **Kanbas**. To implement navigation we'll need to install the **React Router** library from the command line as shown below. Run the command from the root of the project.

```
npm install react-router
```

The React Router library can be used to implement navigation in all kinds of devices including Web applications, mobile, and desktop. To implement navigation in Web application, also install the **React Router DOM** library as follows:

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

Once the library has fully downloaded and installed, let's use the **HashRouter** to implement navigation as shown below. The **HashRouter** 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 **HashRouter** tag, e.g., **HelloWorld**, **Labs** and **Kanbas**.

```tsx
src/App.tsx

import Labs from "./Labs";
import HelloWorld from "./Labs/a3/HelloWorld";
import Kanbas from "./Kanbas";
import {HashRouter} from "react-router-dom";
function App() {
  return (
      <HashRouter>
          <div>
              <HelloWorld/>
              <Labs/>
              <Kanbas/>
          </div>
      </HashRouter>
);}
```

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.

```tsx
src/App.tsx

import Labs from "./Labs";
import HelloWorld from "./Labs/a3/HelloWorld";
import Kanbas from "./Kanbas";
import {HashRouter} from "react-router-dom";
import {Routes, Route, Navigate} from "react-router";
function App() {
  return (
    <HashRouter>
      <div>
        <Routes>
          <Route path="/Labs/*"   element={<Labs/>}/>
          <Route path="/Kanbas/*" element={<Kanbas/>}/>
          <Route path="/hello"    element={<HelloWorld/>}/>
        </Routes>
      </div>
    </HashRouter>
);}
```

Browser

http://localhost:3000/#/hello
Hello World!

http://localhost:3000/#/Labs

Assignment 3

http://localhost:3000/#/Kanbas

Kanbas

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 **Hello World!** message appears. Then confirm navigating to http://localhost:3000/#/Labs displays **Assignment 3**. Then confirm navigating to http://localhost:3000/#/Kanbas displays **Kanbas**. **Note**: you might need to rename the folder **public/kanbas/** to something else like **public/kanbas-old/** since the browser might get confused and render last assignment's **Kanbas** implementation instead of the new **Kanbas** component.

We can declare the **Lab** component as the default landing screen by declaring a route mapped to the root context ("/") that automatically navigates to the **Labs** component. Refresh the browser and confirm that the current assignment component is now the default screen.

src/App.tsx	Browser

``tsx
<HashRouter>
 	
    <Routes>
      <Route path="/"   element={<Navigate to="/Labs"/>}/>
      <Route path="/Labs/*"   element={<Labs/>}/>
      <Route path="/kanbas/*" element={<Kanbas/>}/>
      <Route path="/hello"   element={<HelloWorld/>}/>
    </Routes>
 	
</HashRouter>
``
	http://localhost:3000/
	
	# Assignment 3

2.3.1 Navigating with links in SPAs

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

src/Labs/index.tsx	src/Labs/a3/HelloWorld.tsx	src/Kanbas/index.tsx

``tsx
import {Link} from "react-router-dom";
import Assignment3 from "./a3";
function Labs() {
 return(
 	
   <Link to="/Labs/a3">A3</Link> |
   <Link to="/Kanbas">Kanbas</Link> |
   <Link to="/hello">Hello</Link> |
   <Assignment3/>
 	
)
}
``
	``tsx
	import {Link} from "react-router-dom";
	
	function HelloWorld() {
	 return(
	
	   <Link to="/Labs/a3">A3</Link> |
	   <Link to="/Kanbas">Kanbas</Link> |
	   <Link to="/hello">Hello</Link> |
	    Hello World!
	
	)
	};
	``
		``tsx
		import {Link} from "react-router-dom";
		
		function Kanbas() {
		 return(
		
		   <Link to="/Labs/a3">A3</Link> |
		   <Link to="/Kanbas">Kanbas</Link> |
		   <Link to="/hello">Hello</Link> |
		    Kanbas
		
		)
		}
		``

A3 |Hello |Kanbas # Assignment 3	A3 |Hello |Kanbas # Hello World!	A3 |Hello |Kanbas # Kanbas

2.3.2 Implementing a Navigation component

The navigation links in the three components, **Labs**, **HelloWorld**, and **Kanbas**, would be best implemented as a reusable component as shown below.

src/Nav.tsx

``tsx
import { Link } from "react-router-dom";
function Nav() {
 return (
   <nav className="nav nav-tabs mt-2">
     <Link className="nav-link" to="/Labs/a3">A3</Link>
     <Link className="nav-link" to="/Kanbas">Kanbas</Link>
     <Link className="nav-link" to="/hello">Hello</Link>
   </nav>
 );
}
export default Nav;
``

The component can then be imported into the **HelloWorld**, **Labs**, and **Kanbas** component as shown below. Replace the links with **Nav**, reload your application and confirm the navigation still works.

```
Labs/index.tsx

import Assignment3 from "./a3";
import Nav from "../Nav";
function Labs() {
 return (
   <div>
     <Nav/>
     <Assignment3/>
   </div>
 );
}
export default Labs;
```

```
Labs/a3/HelloWorld.tsx

import Nav from "../../Nav";
function HelloWorld() {
 return (
   <div>
     <Nav/>
     <h1>Hello World!</h1>
   </div>
 );
}
export default HelloWorld;
```

```
Kanbas/index.tsx

import Nav from "../Nav";
function Kanbas() {
 return (
   <div>
     <Nav/>
     <h1>Kanbas</h1>
   </div>
 )
}
export default Kanbas
```

A3   Hello   Kanbas

Assignment 3

A3   Hello   Kanbas

Hello World!

A3   Hello   Kanbas

Kanbas

2.3.3 Encoding Path Parameters

We can encode data in the path to pass parameters between screens. We can parse parameters from the path using the **useParams** React.js hook. The **Add** component below is parsing parameters **a** and **b** from the path and calculating the arithmetic addition of the parameters.

Path Parameters

1 + 2

3 + 4

Add Path Parameters

1 + 2 = 3

```
src/Labs/a3/routing/Add.tsx

import React from "react";
import { useParams } from "react-router-dom";
function Add() {
  const { a, b } = useParams();
  return (
    <div> <h2>Add Path Parameters</h2>
      {a} + {b} = {parseInt(a as string) + parseInt(b as string)}
    </div>
  );
}
export default Add;
```

Path parameter names such as **a** and **b**, are declared in the **path** attribute of the **Route** component. For instance the **Route** component below uses the **colon** character to declare parameters **:a** and **:b**. The first link encodes values 1 and 2 for parameters **a** and **b**, wheres as the second link encodes values 3 and 4 for parameters **a** and **b**. Import **PathParameters** component in your **Assignment3** component and confirm clicking the first link, the URL matches the **Route** and renders the **Add** component with parameters **a=1** and **b=2** and so it renders **1 + 2 = 3**. Confirm clicking the second link sets **a=3** and **b=4** and so it renders **3 + 4 = 7**.

Path Parameters

1 + 2

3 + 4

Add Path Parameters

3 + 4 = 7

```
src/Labs/a3/routing/PathParameters.tsx

import { Routes, Route, Link } from "react-router-dom";
import Add from "./Add";
function PathParameters() {
  return (
    <div>
      <h2>Path Parameters</h2>
      <Link to="/Labs/a3/add/1/2">1 + 2</Link> <br />
```

```
        <Link to="/Labs/a3/add/3/4">3 + 4</Link>
        <Routes>
          <Route path="a3/add/:a/:b" element={<Add />} />
        </Routes>
      </div>
  );
}
export default PathParameters;
```

2.3.4 Working with Location

The **useLocation()** hook returns several properties related to the current URL. The **pathname** property contains the URL itself. We can use the **pathname** to drive UI logic such as highlighting, showing or hiding content based on the URL. The example below destructs the **pathname** from **useLocation()** and then checks to see if the URL contains either **a3**, **hello**, or **Kanbas** to then add the **active** class to highlight the correct tab. Confirm that the correct tab highlights when you click each of the tabs.

src/Nav.tsx

```
import { Link, useLocation } from "react-router-dom";
function Nav() {
  const { pathname } = useLocation();
  return (
    <nav className="nav nav-tabs mt-2">
      <Link to="/Labs/a3"
            className={`nav-link ${pathname.includes("a3") ? "active" : ""}`}>A3</Link>
      <Link to="/Kanbas"
            className={`nav-link ${pathname.includes("Kanbas") ? "active" : ""}`}>Kanbas</Link>
      <Link to="/hello"
            className={`nav-link ${pathname.includes("hello") ? "active" : ""}`}>Hello</Link>
    </nav>
);}
```

2.4 Dynamically Styling React Applications

React.js can generate content dynamically based on algorithms written in JavaScript. We can also dynamically style the content by programmatically controlling the classes and styles applied to the content. In the next couple of exercises we first learn to work with classes and then with styles. Create a component called **DynamicStyling**, import it in the **Assignment3** component, and then import each of the exercises implemented in the next sections. HERE

2.4.1 Working with HTML classes

Let's start practicing simple things, like classes and styles. Under the **Labs/a3** folder, create another folder called **Classes** and create the following component and styling files.

src/Labs/a3/css/Classes/index.tsx

```
import './index.css';
function 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>
   </div> ) };
export default Classes;
```

src/Labs/a3/Classes/index.css

```
.wd-bg-yellow    { background-color: lightyellow; }
.wd-bg-blue      { background-color: lightblue;   }
.wd-bg-red       { background-color: lightcoral;  }
.wd-bg-green     { background-color: lightgreen;  }
.wd-fg-black     { color: black;                  }
.wd-padding-10px { padding: 10px;                 }
```

From the **Assignment3** component, import the new **Classes** component as shown below. Confirm the new **classes** component renders in the screen as expected.

```tsx
Labs/a3/index.tsx
import JavaScript from "./JavaScript";
import PathParameters from "./PathParameters";
import Classes from "./Classes";
function Assignment3() {
 return (
    <div>
      <h1>Assignment 3</h1>
      <Classes/>
      <PathParameters/>
      <JavaScript/>
    </div>
);}
```

Classes

Yellow background

Blue background

Red background

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.

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

Classes

Dynamic Blue background

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.

```tsx
Classes/index.tsx
function Classes() {
 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>); }
```

Classes

Dangerous background

Dynamic Blue background

2.5 Working with the HTML Style attribute

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, implement the **Styles** component below in a new directory **Labs/a3/Styles** and then import it into the **Assignment3** component as shown below. The **Styles** component (**Styles/index.tsx**) 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

Styles

Yellow background

Red background

Blue background

weird syntax of double curly brackets as shown below. Also note that the **Styles** component is implemented using the new arrow function syntax. Refresh the browser and confirm the browser renders as expected. Note we use **background-color** instead of **backgroundColor**.

Labs/a3/css/Styles/index.tsx	Labs/a3/index.tsx

```tsx
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>
       <h2>Styles</h2>
       <div style={{ "backgroundColor": "lightyellow",
          "color": "black", padding: "10px" }}>
         Yellow background</div>
       <div style={ bgRed }> Red background </div>
       <div style={ bgBlue }>Blue background</div>
    </div>
);};
```

```tsx
import JavaScript from "./JavaScript";
import PathParameters from "./PathParameters";
import Classes from "./Classes";
import Styles from "./Styles";
function Assignment3() {
 return (
    <div>
       <h1>Assignment 3</h1>
       <Styles/>
       <Classes/>
       <PathParameters/>
       <JavaScript/>
    </div>
 )
}
export default Assignment3;
```

2.6 Generating 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/Labs/a3/ConditionalOutput/ConditionalOutputIfElse.tsx** with the following code.

src/Labs/a3/ConditionalOutput/ConditionalOutputIfElse.tsx

```tsx
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 we can achieve the same thing by including the conditional content in a boolean expression that short circuits the content if its false, or evaluates the expression if it's true. Implement the equivalent component below in **src/Labs/a3/ConditionalOutput/ConditionalOutputInline.tsx**.

Assignment 3
Welcome If Else
Please login Inline

src/Labs/a3/ConditionalOutput/ConditionalOutputInline.tsx

```tsx
const ConditionalOutputInline = () => {
 const loggedIn = false;
 return (
    <>
       { loggedIn && <h2>Welcome Inline</h2>      }
       {!loggedIn && <h2>Please login Inline</h2> }
    </>
 );
};
export default ConditionalOutputInline;
```

Merge both components into a single component **ConditionalOutputIfElse** as shown below and then import the new component into the **Assignment3** component. Confirm all components render as expected.

Labs/a3/ConditionalOutput/index.tsx	*Labs/a3/index.tsx*

``tsx
import React from "react";
import ConditionalOutputIfElse
  from "./ConditionalOutputIfElse";
import ConditionalOutputInline
  from "./ConditionalOutputInline";
const ConditionalOutput = () => {
 return(
    <>
      <ConditionalOutputIfElse/>
      <ConditionalOutputInline/>
    </>
  );
};
export default ConditionalOutput;
``
	``tsx
	import Classes from "./Classes";
	import Styles from "./Styles";
	import JavaScript from "./JavaScript";
	import PathParameters from "./PathParameters";
	import ConditionalOutput from "./ConditionalOutput";
	function Assignment3() {
	 return (
	
	       Assignment 3
	      <ConditionalOutput/>
	      ...
	      <JavaScript/>
	
	  );}
	``

2.7 Child Components

In previous exercises we've demonstrated components nested inside other components. For instance the App component nests the **Labs**, **Kanbas** and **HelloWorld** components. Then the **Labs** component nests the **Assignment3** component. **React** encourages this strategy to break up complex user interfaces into smaller, more maneageable pieces. Another nesting strategy is to wrap nested components using the **children** property as follows:

``tsx
import { ReactNode } from "react";
function Highlight({ children }: { children: ReactNode }) {
  return (

      {children}

  );
}
export default Highlight;
``
	`//`

In the above example, the **Highlight** component is receiving the **children** property as a property and then rendering it in the return statement. The **children** property is a special property that contains the **body** of component when using its opening and closing tags as shown below. Implement the **Highlight** component, import it in the **Assignment3**, and confirm it works.

src/Labs/a3/index.tsx

```tsx
...
import Highlight from "./Highlight";

function Assignment3() {
 return (
    <div>
      <h1>Assignment 3</h1>
      ...
      <Highlight>
        Lorem ipsum dolor sit amet consectetur adipisicing elit. Suscipitratione eaque illo minus cum, saepe totam
        vel nihil repellat nemo explicabo excepturi consectetur. Modi omnis minus sequi maiores, provident voluptates.
      </Highlight>
    </div>
  );}
```

2.8 Component attributes and properties

React components can be parameterized by using the familiar HTML attribute syntax, which are passed as properties in the component's function definition. The following **Add** component can receive properties **a** and **b** deconstructed from the attributes of its equivalent HTML attributes syntax. Implement the **Add** component and confirm that passing it a=3 and b=4 results in a + b = 3.

```tsx
function Add({ a, b }: { a: number; b: number }) {
  return (
    <>
      <h3>Add</h3>a = {a}
      <br />b = {b}
      <br />a + b = {a + b}
    </>
  );
}

export default Add;
```

```tsx
import Add from "./Add";
function Assignment3() {
  return (
    <div className="container">
      <h1>Assignment 3</h1>
      ...
      <Add a={3} b={4} />
    </div>
  );
}
export default Assignment3;
```

2.9 Implementing a simple ToDo List using React.js

Let's bring together several of the concepts covered so far and implement a **Todo** list application that renders a list of todos dynamically using React.js. In a new directory **src/Labs/a3/todo**, implement the **TodoItem** component in a **TodoItem.tsx** file as shown below. Import the component into the **Assignment3** component and confirm that it renders as shown.

src/Labs/a3/todos/TodoItem.tsx

```tsx
const TodoItem = ( { todo = { done: true, title: 'Buy milk',
                               status: 'COMPLETED' } }) => {
  return (
    <li className="list-group-item">
      <input type="checkbox" className="me-2"
             defaultChecked={todo.done}/>
      {todo.title} ({todo.status})
    </li>
  );
}
export default TodoItem;
```

Assignment 3
☑ Buy milk(COMPLETED)

Create a JSON file **todos.json** that contains an array of todos as shown below.

src/Labs/a3/todos/todos.json

```json
[
  { "title": "Buy milk",        "status": "CANCELED",    "done": true   },
  { "title": "Pickup the kids", "status": "IN PROGRESS", "done": false  },
  { "title": "Walk the dog",    "status": "DEFERRED",    "done": false  }
]
```

Now let's implement a TodoList component that renders the array of todos as shown below. Import the component in **Labs/a3/index.tsx**, refresh the browser, and confirm the **TodoList** renders a list of checkboxes and todo items.

/src/Labs/a3/todos/TodoList.tsx

```
import TodoItem from "./TodoItem";
import todos from "./todos.json";
const TodoList = () => {
 return(
   <>
     <h3>Todo List</h3>
     <ul className="list-group">
       { todos.map(todo => {
           return(<TodoItem todo={todo}/>);
         })}
     </ul>
   </>
 );
}
export default TodoList;
```

Todo List

- ☑ Buy milk(CANCELED)
- ☐ Pickup the kids(IN PROGRESS)
- ☐ Walk the dog(DEFERRED)

3 Implementing the Kanbas Application with React.js

In previous assignments we used **HTML** and **CSS** to create various screens of the **Kanbas** application. In this section we're going to reuse the **HTML** and **CSS** content to create equivalent React.js components. Let's start by creating placeholders for the left and right hand side of the application using **Flex** as shown below.

```
/src/Kanbas/index.tsx
```

```
function Kanbas() {
  return (
    <div className="d-flex">
      <div>
        <h1>Kanbas Navigation</h1>
      </div>
      <div style={{ flexGrow: 1 }}>
        <h1>Account</h1>
        <h1>Dashboard</h1>
        <h1>Courses</h1>
      </div>
    </div>
);}
export default Kanbas;
```

3.1 Implementing Kanbas Navigation as a React.js component

Based on your **Kanbas Navigation** HTML and CSS implemented in earlier assignments, e.g, **/public/Kanbas/ Navigation/index.html** and **index.css**, create a React.js component called **KanbasNavigation** that renders an array of links as shown here on the right. Use the **Link** component to implement the hyperlinks and set the **to** attributes to the following paths

- /Kanbas/Account
- /Kanbas/Dashboard
- /Kanbas/Courses
- /Kanbas/Calendar

Here's an example of how you can implement the **Kanbas Navigation** component

```
/src/Kanbas/Navigation/index.tsx
```

```
import { Link, useLocation } from "react-router-dom";
import "./index.css";
import { FaTachometerAlt, FaRegUserCircle, FaBook, FaRegCalendarAlt } from "react-icons/fa";
function KanbasNavigation() {
  const links = [
    { label: "Account",   icon: <FaRegUserCircle className="fs-2" />  },
    { label: "Dashboard", icon: <FaTachometerAlt className="fs-2" />  },
    { label: "Courses",   icon: <FaBook className="fs-2" />           },
    { label: "Calendar",  icon: <FaRegCalendarAlt className="fs-2" /> },
  ];
  const { pathname } = useLocation();
  return (
    <ul className="wd-kanbas-navigation">
      {links.map((link, index) => (
        <li key={index} className={pathname.includes(link.label) ? "wd-active" : ""}>
          <Link to={`/Kanbas/${link.label}`}> {link.icon} {link.label} </Link>
        </li>
      ))}
    </ul>
  );
}
export default KanbasNavigation;
```
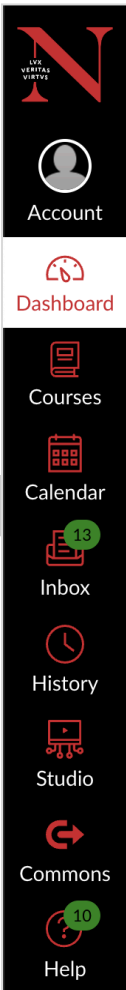
Render the **Kanbas Navigation** component on the left of the **Kanbas** application. Here's an example of how you could render the **Kanbas Navigation** component on the left of the **Kanbas** application. Import **CSS** files as needed (not shown).

```
/src/Kanbas/index.tsx
```

```tsx
import KanbasNavigation from "./Navigation";
function Kanbas() {
  return (
    <div className="d-flex">
      <KanbasNavigation />
      <div style={{ flexGrow: 1 }}>
        <h1>Account</h1>
        <h1>Dashboard</h1>
        <h1>Courses</h1>
      </div>
    </div>
  );
}
```

3.2 Implement a JSON file as a "Database"

Implement the data file that will contain all the data needed in the **Kanbas** application. We'll start by adding courses in a **courses.json** file under a new **Database** folder. Each course should have the following fields:

- **_id**: a unique identifier for the course
- **name**: the name of the course, e.g., Web Development
- **number**: the number of the course, e.g., CS4550
- **startDate**: the date of the start of the course, e.g., 2023-09-07
- **endDate**: the date of the end of the course, e.g., 2023-12-15

Here's an example of some courses. Feel free to reuse these or make up your own.

```
/src/Kanbas/Database/courses.json
```

```json
[ { "_id": "RS101", "name": "Rocket Propulsion", "number": "RS4550", "startDate": "2023-01-10",
    "endDate": "2023-05-15", "image": "rocket-propulsion.png"  },
  { "_id": "RS102", "name": "Aerodynamics", "number": "RS4560", "startDate": "2023-01-10",
    "endDate": "2023-05-15", "image": "aerodynamics.png"  },
  { "_id": "RS103", "name": "Spacecraft Design", "number": "RS4570", "startDate": "2023-01-10",
    "endDate": "2023-05-15", "image": "spaceship-design.png"  } ]
```

Create a database file that contains all the data needed for the **Kanbas** application. For now we just have the courses, but later sections will add users, assignments, modules, etc.

```
/src/Kanbas/Database/index.tsx
```

```tsx
import courses from "./courses.json";
export {  courses  };
```

3.3 Implementing the Dashboard Screen with React.js
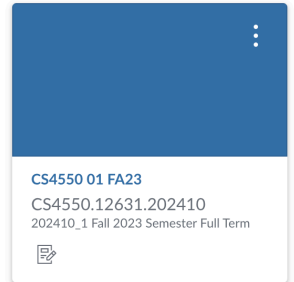
Based on your implementation of the **Dashboard** in previous assignments, implement a React.js **Dashboard** component that renders at least 3 courses of your choice. The component should render as described in previous assignments. The following code is an example of how you can implement the **Dashboard** component dynamically rendering an array of courses. Combine this approach with the layout and styling of the **Dashboard** implemented in earlier assignments.

```tsx
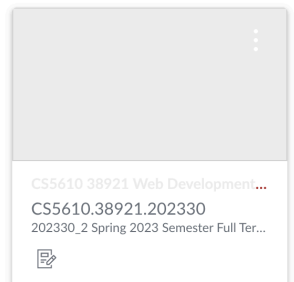import React from "react";
import { Link } from "react-router-dom";
import { courses } from "../Database";
function Dashboard() {
  return (
    <div className="p-4">
      <h1>Dashboard</h1>              <hr />
      <h2>Published Courses (12)</h2> <hr />
      <div className="row">
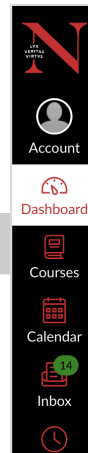        <div className="row row-cols-1 row-cols-md-5 g-4">
          {courses.map((course) => (
            <div key={course._id} className="col" style={{ width: 300 }}>
              <div className="card">
                <img src={`/images/${course.image}`} className="card-img-top"
                  style={{ height: 150 }}/>
                <div className="card-body">
                  <Link className="card-title" to={`/Kanbas/Courses/${course._id}/Home`}
                    style={{ textDecoration: "none", color: "navy", fontWeight: "bold" }}>
                    {course.name} </Link>
                  <p className="card-text">{course.name}</p>
                  <Link to={`/Kanbas/Courses/${course._id}/Home`} className="btn btn-primary">
                    Go </Link>
                </div>
              </div>
            </div>
          ))}
        </div>
      </div>
    </div>
  );
}
export default Dashboard;
```

The *Link* component is used to create the hyperlinks and encoding the course's ID at the end of the path. This can then be used in other components to parse the ID from the URL and display content specific to the selected course. Below is an example of how React routing can be used to navigate to the Dashboard by default. Confirm that Dashboard is the default screen and that the Dashboard link is highlighted as shown here on the right.

```tsx
import KanbasNavigation from "./Navigation";
import { Routes, Route, Navigate } from "react-router-dom";
import Dashboard from "./Dashboard";
function Kanbas() {
  return (
    <div className="d-flex">
      <KanbasNavigation />
      <div style={{ flexGrow: 1 }}>
        <h1>Account</h1>
        <h1>Dashboard</h1>
        <h1>Courses</h1>
        <Routes>
          <Route path="/" element={<Navigate to="Dashboard" />} />
          <Route path="Account" element={<h1>Account</h1>} />
          <Route path="Dashboard" element={<Dashboard />} />
          <Route path="Courses/*" element={<h1>Courses</h1>} />
        </Routes>
      </div>
    </div>
  );
}
```

3.3 Implementing the Courses Screen with React.js

Implement a **Courses** component that can render a course when you select it from the **Dashboard**. For now we'll display the name of the course selected. Later we'll also add the screen we're in like **Home**, **Assignments**, or **Grades**. Below is an example how you can extract the ID of the course from the URL path, then use it to find the corresponding course object from the database object, and then display the course's name.

```
/src/Kanbas/Courses/index.tsx

import { courses } from "../../Kanbas/Database";
import { useParams } from "react-router-dom";
import { HiMiniBars3 } from "react-icons/hi2";
function Courses() {
  const { courseId } = useParams();
  const course = courses.find((course) => course._id === courseId);
  return (
    <div>
      <h1><HiMiniBars3 /> Course {course?.name}</h1>
    </div>
  );
}
export default Courses;
```

≡ CS4550 01 FA23

Below is an example of using React routing to navigate to the **Courses** screen when the URL matches **/Kanbas/Courses/:courseId**. Confirm that selecting a course from the **Dashboard** navigates to the **Courses** screen and displays the correct course name on the right as shown below on the right.

```
src/Kanbas/index.tsx

import { Routes, Route, Navigate } from "react-router-dom";
import Courses from "./Courses";
function Kanbas() {
  return (
    <div className="d-flex">
      <KanbasNavigation />
      <div style={{ flexGrow: 1 }}>
        <Routes>
          <Route path="/" element={<Navigate to="Dashboard" />} />
          <Route path="Account" element={<h1>Account</h1>} />
          <Route path="Dashboard" element={<Dashboard />} />
          <Route path="Courses/:courseId/*" element={<Courses />} />
        </Routes>
      </div>
    </div>
  );
}
```
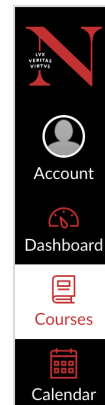
3.4 Implementing the Course Navigation as a React.js component

Based on your implementation of the **Course Navigation** in previous assignments, implement a React.js **CourseNavigation** component that renders an array of links as shown below on the right. Feel free to reuse the HTML and CSS from previous assignments. Below is an example of how **Course Navigation** can be implemented as an array of links. The course's ID is read from the URL to encode it in the links using **useParams()**. The URL **pathname** from **useLocation()** is used to highlight the links.

Home
Modules
Piazza
Zoom Meetings
Assignments
Quizzes
Grades
People

```
/src/Kanbas/Courses/Navigation/index.tsx

import { Link, useLocation } from "react-router-dom";
import "./index.css"; // feel free to use the CSS from previous assignments
function CourseNavigation() {
  const links = ["Home", "Modules", "Piazza", "Grades", "Assignments"];
```

```
    const { pathname } = useLocation();
    return (
      <ul className="wd-navigation">
        {links.map((link, index) => (
          <li key={index} className={pathname.includes(link) ? "wd-active" : ""}>
            <Link to={link}>{link}</Link>
          </li>
        ))}
      </ul>
    );
}
export default CourseNavigation;
```

Add the *Course Navigation* component to the *Courses* screen as shown below. Then add routes to render various screens based on the paths in the *CourseNavigation* component. Reuse HTML and styles implemented in earlier assignments to achieve a similar look and feel as shown below on the right. Confirm that the *Home* link is the default and highlighted, as well as *Home* is displayed on the right side since it matches the URL path. Also, make sure *Home* is displayed in the header after the name of the course as a breadcrumb.

src/Kanbas/Courses/index.tsx

```
import { courses } from "../../Kanbas/Database";
import { Navigate, Route, Routes, useParams } from "react-router-dom";
import { HiMiniBars3 } from "react-icons/hi2";
import CourseNavigation from "./Navigation";

function Courses() {
  const { courseId } = useParams();
  const course = courses.find((course) => course._id === courseId);
  return (
    <div>
      <h1><HiMiniBars3 /> Course {course?.name}</h1>
      <CourseNavigation />
      <div>
        <div
          className="overflow-y-scroll position-fixed bottom-0 end-0"
          style={{ left: "320px", top: "50px" }} >
          <Routes>
            <Route path="/" element={<Navigate to="Home" />} />
            <Route path="Home" element={<h1>Home</h1>} />
            <Route path="Modules" element={<h1>Modules</h1>} />
            <Route path="Piazza" element={<h1>Piazza</h1>} />
            <Route path="Assignments" element={<h1>Assignments</h1>} />
            <Route path="Assignments/:assignmentId" element={<h1>Assignment Editor</h1>} />
            <Route path="Grades" element={<h1>Grades</h1>} />
          </Routes>
        </div>
      </div>
    </div>
  );
}
```
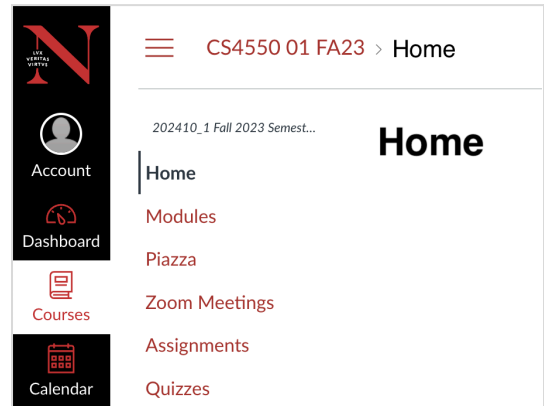
3.5 Implementing the Modules Screen with React.js

Based on the *Home* screen implemented in earlier assignments, create a *Modules* component that renders an array of modules. Each course has a different set of modules. Use the data provided as an example to create a *modules.json* file that contains at least 3 modules of your choice for each course. Include the modules in the *Database* as shown below.

/src/Kanbas/Database/index.tsx

```
import courses from "./courses.json";
import modules from "./modules.json";
export { courses, modules };
```

Below is an example of how you can use the *modules* in the *Database* to render a list of modules. Use useParams() to retrieve the course ID from the URL and then retrieve the corresponding modules for the course. Feel free to reuse any HTML and CSS code from previous assignments to render the list of modules similar to the image below on the right. Use react icons similar to the ones shown.

/src/Kanbas/Courses/Modules/List.tsx

```tsx
import React, { useState } from "react";
import "./index.css";
import { modules } from "../../Database";
import { FaEllipsisV, FaCheckCircle, FaPlusCircle } from "react-icons/fa";
import { useParams } from "react-router";
function ModuleList() {
  const { courseId } = useParams();
  const modulesList = modules.filter((module) => module.course === courseId);
  const [selectedModule, setSelectedModule] = useState(modulesList[0]);
  return (
    <>
      {/* <!-- Add buttons here --> */}
      <ul className="list-group wd-modules">
        {modulesList.map((module) => (
          <li
            className="list-group-item"
            onClick={() => setSelectedModule(module)}>
            <div>
              <FaEllipsisV className="me-2" />
              {module.name}
              <span className="float-end">
                <FaCheckCircle className="text-success" />
                <FaPlusCircle className="ms-2" />
                <FaEllipsisV className="ms-2" />
              </span>
            </div>
            {selectedModule._id === module._id && (
              <ul className="list-group">
                {module.lessons?.map((lesson) => (
                  <li className="list-group-item">
                    <FaEllipsisV className="me-2" />
                    {lesson.name}
                    <span className="float-end">
                      <FaCheckCircle className="text-success" />
                      <FaEllipsisV className="ms-2" />
                    </span>
                  </li>
                ))}
              </ul>
            )}
          </li>
        ))}
      </ul>
    </>
  );
}
export default ModuleList;
```

Create a *Modules* screen that renders a heading and the list of modules as shown below.

src/Kanbas/Courses/Modules/index.tsx

```tsx
import ModuleList from "./List";
function Modules() {
  return (
    <div>
      <h2>Modules</h2>
      <ModuleList />
    </div>
  );
}
export default Modules;
```

Add the **Modules** screen to the **Courses** routes so that when you click the **Modules** link, the **Modules** screen is shown and the **Modules** link is highlighted. Also make sure that **Modules** is shown in the breadcrumb in the heading after the course name as shown below on the right.

src/Kanbas/Courses/index.tsx

```tsx
import { courses } from "../../Kanbas/Database";
import { Navigate, Route, Routes, useParams } from "react-router-dom";
import { HiMiniBars3 } from "react-icons/hi2";
import CourseNavigation from "./CourseNavigation";
import Modules from "./Modules";
function Courses() {
  const { courseId } = useParams();
  const course = courses.find(
    (course) => course._id === courseId);
  return (
    <div>
      <h1><HiMiniBars3 /> Course {course?.name}</h1>
      <CourseNavigation />
      <div>
        <div>
          <Routes>
            <Route path="/" element={<Navigate to="Home" />} />
            <Route path="Home" element={<h1>Home</h1>} />
            <Route path="Modules" element={<Modules/>} />
            <Route path="Assignments"
                   element={<h1>Assignments</h1>} />
            <Route path="Assignments/:assignmentId"
                   element={<h1>Assignment Editor</h1>}/>
            <Route path="Grades" element={<h1>Grades</h1>} />
          </Routes>
        </div>
      </div>
    </div>
  );}
```

3.6 Implementing the Home Screen with React.js

Similar to the **Modules** screen, create a **Home** screen that renders the list of modules, but also the **Course Status** side bar on the right as shown here on the right. Here's an example of how to implement the **Home** screen rendering the list of modules and a placeholder for the **Status**. Reuse HTML and CSS from previous assignments so that the **Home** screen looks similar to the image shown here on the right.

/src/Kanbas/Courses/Home/index.tsx

```tsx
import ModuleList from "../Modules/List";

function Home() {
  return (
    <div>
      <h2>Home</h2>
      <ModuleList />
      <h2>Status</h2>
    </div>
  );
}
export default Home;
```

Add the **Home** screen to the **Courses** as a route so that when you click on the **Home** link the **Home** screen is displayed and the **Home** link is highlighted.

```
src/Kanbas/Courses/index.tsx
```

```tsx
import { courses } from "../../Kanbas/Database";
import { Navigate, Route, Routes, useParams } from "react-router-dom";
import { HiMiniBars3 } from "react-icons/hi2";
import CourseNavigation from "./CourseNavigation";
import Modules from "./Modules";
import Home from "./Home";

function Courses() {
  const { courseId } = useParams();
  const course = courses.find((course) => course._id === courseId);
  return (
    <div>
        <h1><HiMiniBars3 /> Course {course?.name}</h1>
        <CourseNavigation />
        <div>
          <div className="overflow-y-scroll position-fixed bottom-0 end-0">
            <Routes>
              <Route path="/" element={<Navigate to="Home" />} />
              <Route path="Home" element={<Home/>} />
              <Route path="Modules" element={<Modules/>} />
              <Route path="Assignments" element={<h1>Assignments</h1>} />
              <Route path="Assignments/:assignmentId" element={<h1>Assignment Editor</h1>}/>
              <Route path="Grades" element={<h1>Grades</h1>} />
            </Routes>
          </div>
        </div>
    </div>
);}
```

3.7 Implementing the Assignments Screen with React.js

Based on your implementation of the **Assignments** screen in previous assignments, create a React.js **Assignments** complement that renders an array of assignments. Each course has a different set of assignments. Use the data provided as an example to create an **assignments.json** file that contains at least 3 assignments of your choice for each course. Include the **assignments** in the **Database** as shown below.

```
src/Kanbas/Database/index.tsx
```

```tsx
import courses from "./courses.json";
import modules from "./modules.json";
import assignments from "./assignments.json";
export default {
  courses, modules, assignments,
};
```


Here's an example of rendering the assignments for the selected course. The **useParams()** hook is used to parse the course's ID and then find all the assignments for that course from the database's **assignments** array. The assignments are rendered as links that encode the course's ID and the assignment's ID in the URL's path. This will be used by a router to render the corresponding assignment in the **AssignmentEditor** screen. Use the code below as an example and feel free to reuse any HTML and CSS code from previous assignments to render the screen similar to the image shown here on the right.

/src/Kanbas/Courses/Assignments/index.tsx

```tsx
import React from "react";
import { FaCheckCircle, FaEllipsisV, FaPlusCircle } from "react-icons/fa";
import { Link, useParams } from "react-router-dom";
import { assignments } from "../../Database";
function Assignments() {
  const { courseId } = useParams();
  const assignmentList = assignments.filter(
    (assignment) => assignment.course === courseId);
  return (
    <>
      {<!-- Add buttons and other fields here -->}
      <ul className="list-group wd-modules">
        <li className="list-group-item">
          <div>
            <FaEllipsisV className="me-2" /> ASSIGNMENTS
            <span className="float-end">
              <FaCheckCircle className="text-success" />
              <FaPlusCircle className="ms-2" /><FaEllipsisV className="ms-2" />
            </span>
          </div>
          <ul className="list-group">
            {assignmentList.map((assignment) => (
              <li className="list-group-item">
                <FaEllipsisV className="me-2" />
                <Link
                  to={`/Kanbas/Courses/${courseId}/Assignments/${assignment._id}`}>{assignment.title}</Link>
                <span className="float-end">
                  <FaCheckCircle className="text-success" /><FaEllipsisV className="ms-2" /></span>
              </li>))}
          </ul>
        </li>
      </ul>
    </>
);}
export default Assignments;
```

In the **Courses**, render the **Assignments** screen when you click on the **Assignments** link. Use the code below as an example.

src/Kanbas/Courses/index.tsx

```tsx
import { Navigate, Route, Routes, useParams } from "react-router-dom";
import CourseNavigation from "./CourseNavigation";
import { HiMiniBars3 } from "react-icons/hi2";
import Modules from "./Modules";
import Home from "./Home";
import Assignments from "./Assignments";
function Courses() {
  const { courseId } = useParams();
  const course = courses.find((course) => course._id === courseId);
  return (
    <div>
      <h1><HiMiniBars3 /> Course {course?.name}</h1>
      <CourseNavigation />
      <div>
        <div className="overflow-y-scroll position-fixed bottom-0 end-0">
          <Routes>
            <Route path="/" element={<Navigate to="Home" />} />
            <Route path="Home" element={<Home/>} />
            <Route path="Modules" element={<Modules/>} />
            <Route path="Assignments" element={<Assignments/>} />
            <Route path="Assignments/:assignmentId" element={<h1>Assignment Editor</h1>}/>
            <Route path="Grades" element={<h1>Grades</h1>} />
          </Routes>
        </div>
      </div>
    </div>
);}
```

3.8 Implementing the Assignment Editor Screen with React.js (graduates only)

Based on the **Assignment Editor** screen implemented in earlier assignments, create a React.js **AssignmentEditor** component that renders the information for the assignment selected in the **Assignments** screen. Below is an example of the **AssignmentEditor** using **useParams()** to parse the assignment's and course's ID from the URL and retrieving the assignment from the database object. The course's ID is used to navigate back to the **Assignments** screen after clicking the **Cancel** or **Save** buttons. The assignment's name is displayed in an input field ready for editing. The **Cancel** button is implemented as a **Link** so that it just navigates back to the **Assignments** screen for the current course. The **Save** button instead is handled by the **handleSave** function which could be used to actually save the changes in a later assignment. It then also navigates to the **Assignments** screen.

/src/Kanbas/Courses/Assignments/Editor/index.tsx

```tsx
import React from "react";
import { useNavigate, useParams, Link } from "react-router-dom";
import { assignments } from "../../../Database";
function AssignmentEditor() {
  const { assignmentId } = useParams();
  const assignment = assignments.find(
    (assignment) => assignment._id === assignmentId);
  const { courseId } = useParams();
  const navigate = useNavigate();
  const handleSave = () => {
    console.log("Actually saving assignment TBD in later assignments");
    navigate(`/Kanbas/Courses/${courseId}/Assignments`);
  };
  return (
    <div>
      <h2>Assignment Name</h2>
      <input value={assignment?.title}
             className="form-control mb-2" />
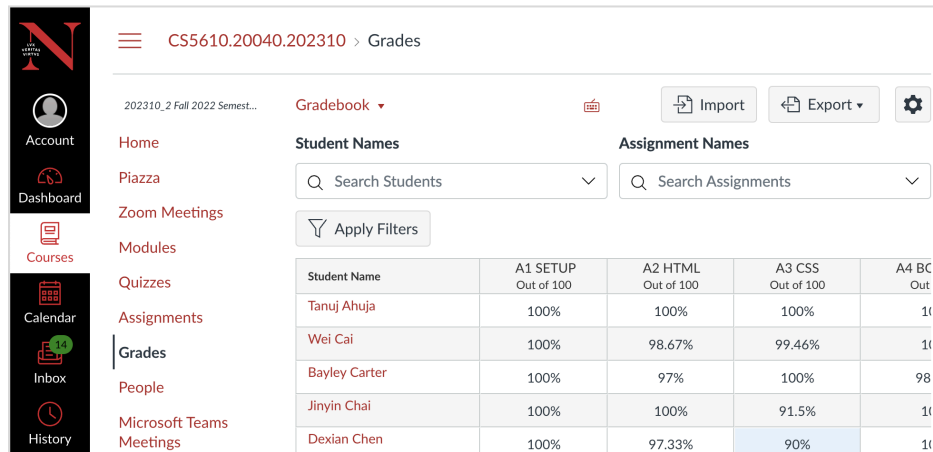      <button onClick={handleSave} className="btn btn-success ms-2 float-end">
        Save
      </button>
      <Link to={`/Kanbas/Courses/${courseId}/Assignments`}
            className="btn btn-danger float-end">
        Cancel
      </Link>
    </div>
  );
}
export default AssignmentEditor;
```

Add a route to **Courses** so that when you click on an assignment in the **Assignments** screen, the **AssignmentEditor** screen renders the correct assignment based on the **assignmentId** encoded in the path. Use the code below as guide. Feel free to reuse any HTML and CSS code from previous assignments. The example here on the right only renders the name of the course, but feel free to render additional assignment fields, but they are not required.

---

src/Kanbas/Courses/index.tsx

```tsx
import { courses } from "../../Kanbas/Database";
import { Navigate, Route, Routes, useParams } from "react-router-dom";
import CourseNavigation from "./CourseNavigation";
import { HiMiniBars3 } from "react-icons/hi2";
import Modules from "./Modules";
import Home from "./Home";
```

```
import Assignments from "./Assignments";
import AssignmentEditor from "./Assignments/Editor";

function Courses() {
  const { courseId } = useParams();
  const course = courses.find(
    (course) => course._id === courseId);
  return (
    <div>
      <h1><HiMiniBars3 /> Course {course?.name}</h1>
      <CourseNavigation />
      <div>
        <div className="overflow-y-scroll position-fixed bottom-0 end-0">
          <Routes>
            <Route path="/" element={<Navigate to="Home" />} />
            <Route path="Home" element={<Home/>} />
            <Route path="Modules" element={<Modules/>} />
            <Route path="Assignments" element={<Assignments/>} />
            <Route path="Assignments/:assignmentId" element={<AssignmentEditor/>}/>
            <Route path="Grades" element={<h1>Grades</h1>} />
          </Routes>
        </div>
      </div>
    </div>
  );
}
```

3.9 Grades Screen (graduates only)

Based on the **Grades** screen implemented in previous assignments, create a React.js **Grades** component that renders the students in current course and the grades they got in the assignments for that course. Use the data provided as an example to create an **users.json** file that contains at least 10 users of your choice for each course. Also create an **enrollments.json** file that contains enrollment data establishing which students are enrolled in which course as shown below.

src/Kanbas/Database/enrollments.json

```
[ { "_id": "1", "user": "121", "course": "RS101" },
  { "_id": "2", "user": "122", "course": "RS101" },
  { "_id": "3", "user": "123", "course": "RS101" },
  { "_id": "4", "user": "124", "course": "RS102" },
  { "_id": "5", "user": "125", "course": "RS102" },
  { "_id": "6", "user": "126", "course": "RS102" },
  { "_id": "7", "user": "121", "course": "RS103" },
  { "_id": "8", "user": "124", "course": "RS103" },
  { "_id": "9", "user": "127", "course": "RS103" } ]
```

Use the data provided as an example to create an **grades.json** file that contains at least 3 grades of your choice for each student. Include the **users**, **grades** and **enrollments** in the **Database** as shown below.

src/Kanbas/Database/index.tsx

```
import courses from "./courses.json";
import modules from "./modules.json";
import assignments from "./assignments.json";
import users from "./users.json";
```

```
import enrollments from "./enrollments.json";
import grades from "./grades.json";
export default {
  courses, modules, assignments,
  users, enrollments, grades
};
```

Below is an example of the **Grades** screen parsing the current course's ID from the URL and then retrieving the **assignments** and **enrollments** for the current course. We iterate over the **assignments** array to render the assignment titles as headings.

src/Kanbas/Courses/Grades/index.tsx

```
import { assignments, enrollments, grades, users } from "../../Database";
import { useParams } from "react-router-dom";
function Grades() {
  const { courseId } = useParams();
  const as = assignments.filter((assignment) => assignment.course === courseId);
  const es = enrollments.filter((enrollment) => enrollment.course === courseId);
  return (
    <div>
      <h1>Grades</h1>
      <div className="table-responsive">
        <table className="table">
          <thead>
            <th>Student Name</th>
            {as.map((assignment) => (<th>{assignment.title}</th>))}
          </thead>
```

Then we iterate over the **enrollments** which include the users in this course. For each **enrollment** we find the corresponding **user** object so we can render their first and last names. For each **enrollment**, we also iterate over the **assignments** for this course looking for the grade in the assignment that matches for this student.

src/Kanbas/Courses/Grades/index.tsx (continued)

```
        <tbody>
          {es.map((enrollment) => {
            const user = users.find((user) => user._id === enrollment.user);
            return (
              <tr>
                <td>{user?.firstName} {user?.lastName}</td>
                {assignments.map((assignment) => {
                  const grade = grades.find(
                    (grade) => grade.student === enrollment.user && grade.assignment === assignment._id);
                    return (<td>{grade?.grade || ""}</td>);})}
              </tr>);
          })}
        </tbody></table>
    </div></div>);
}
export default Grades;
```

Add a route to **Course** so that clicking **Grades** link navigates to the **Grades** screen as shown below.

src/Kanbas/Courses/index.tsx

```
import { courses } from "../../Kanbas/Database";
import { Navigate, Route, Routes, useParams } from "react-router-dom";
import CourseNavigation from "./CourseNavigation";
import { HiMiniBars3 } from "react-icons/hi2";
import Modules from "./Modules";
import Home from "./Home";
import Assignments from "./Assignments";
import AssignmentEditor from "./Assignments/Editor";
import Grades from "./Grades";
```

```
function Courses() {
  const { courseId } = useParams();
  const course = courses.find((course) => course._id === courseId);
  return (
    <div>
      <h1><HiMiniBars3 /> Course {course?.name}</h1>
      <CourseNavigation />
      <div>
        <div className="overflow-y-scroll position-fixed bottom-0 end-0">
          <Routes>
            <Route path="/" element={<Navigate to="Home" />} />
            <Route path="Home" element={<Home/>} />
            <Route path="Modules" element={<Modules/>} />
            <Route path="Assignments" element={<Assignments/>} />
            <Route path="Assignments/:assignmentId" element={<AssignmentEditor/>}/>
            <Route path="Grades" element={<Grades />} />
          </Routes>
        </div>
      </div>
    </div>
  );
}
export default Courses;
```

4 Deliverables

As a deliverable, make sure you complete the **Labs** and **Kanbas** sections of this assignment. All your work must be done in a branch called **a3**. 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 **a3***

```
git  checkout  -b  a3
# do all your work
```

Do all your work, e.g., **Labs** exercises and **Kanbas**

Add, commit and push the new branch

```
git  add  .
git  commit  -am  "JavaScript and React.js Assignment 3"
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 **a3** branch deployed to on Netlify
2. The link to your new branch in GitHub.