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***). We've already been using JavaScript in previous assignments, but mostly for creating static Web pages. In this assignment we're going to learn how to use JavaScript to handle user input, manipulate data structures, parameterize components, and create data driven user interfaces.

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 ***VS Code*** or ***IntelliJ*** 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 as shown below.

public/index.html

```
<html>
  ...
  <body>
    <div id="root"></div>
  </body>
</html>
```

The ***React.js*** application is implemented in ***src/index.tsx*** importing ***React*** and ***ReactDOM*** libraries as shown below. ***ReactDOM*** uses ***document.getElementById('root')*** to retrieve a reference to the ***DOM*** element declared in ***index.html***, invokes the ***App*** component and appends its output to the ***DIV*** element whose ***ID*** is ***root***. The ***src/App.tsx*** is the entry point of the ***React.js*** application we're building and it contained code generated by the ***create-react-app*** tool. In previous assignment we replaced the content of ***App.tsx*** with ***Labs*** and ***Kanbas***, our own version of the ***Canvas*** OLMS.

src/index.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') as HTMLElement);
root.render(
 <React.StrictMode>
   <App />
 </React.StrictMode>
);
```

2.2 JavaScript Basics

In the following exercises, we'll learn about the ***JavaScript*** programming language. We'll create a component for each exercise and import them into the ***Lab3*** component. Create ***src/Labs/Lab3/index.tsx*** and import it from the ***Labs*** component as shown below.

src/Labs/Lab3/index.tsx

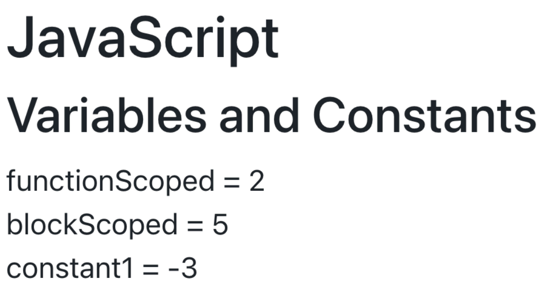
```
export default function Lab3() {
  return (
    <div id="wd-lab3">
      <h3>Lab 3</h3>
    </div>
  );
}
```

src/Labs/index.tsx

```
import Lab3 from "./Lab3";
export default function Labs() {
  return (
    <div className="p-3">
      <h1>Labs</h1>
      <TOC/>
      <Routes>
        <Route path="/" element={<Navigate to="Lab1" />} />
        <Route path="Lab1" element={<Lab1 />} />
        <Route path="Lab2" element={<Lab2 />} />
        <Route path="Lab3/*" element={<Lab3 />} />
      </Routes>
    </div>
);}
```

2.2.1 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 ***Lab3*** 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 ***Lab3*** 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/Lab3/VariablesAndConstants.tsx

```
export default function VariablesAndConstants() {
 var functionScoped = 2;
 let blockScoped = 5;
 const constant1 = functionScoped - blockScoped;
 return(
   <div id="wd-variables-and-constants">
     <h4>Variables and Constants</h4>
     functionScoped = { functionScoped }<br/>
     blockScoped = { blockScoped }<br/>
     constant1 = { constant1 }<hr/>
   </div>
);}
```

src/Labs/Lab3/index.tsx

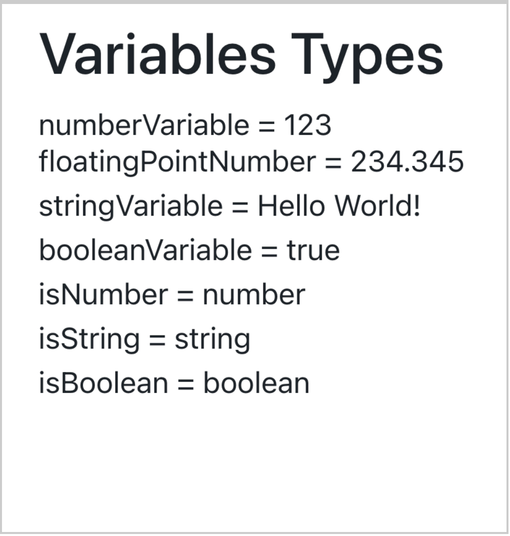
```
import VariablesAndConstants from "./VariablesAndConstants";

export default function Lab3() {
  return(
    <div id="wd-lab3">
      <h3>Lab 3</h3>
      <VariablesAndConstants/>
    </div>
  );
}
```

2.2.2 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 at the bottom of the ***Lab3*** component. Confirm that the browser renders as shown here 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/Lab3/VariableTypes.tsx — *Browser*

```
export default 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 id="wd-variable-types">
      <h4>Variables Types</h4>
      numberVariable = { numberVariable }<br/>
      floatingPointNumber = { floatingPointNumber }<br/>
      stringVariable = { stringVariable }<br/>
      booleanVariable = { booleanVariable + "" }<br/>
      isNumber = { isNumber }<br/>
      isString = { isString }<br/>
      isBoolean = { isBoolean }<hr/>
    </div>
);}
```



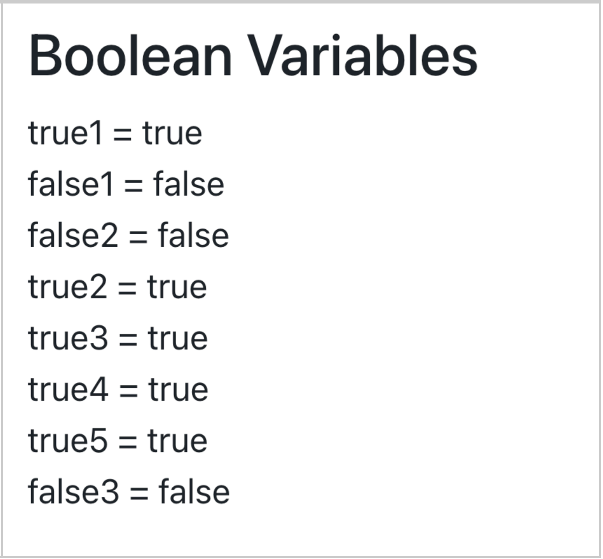
```
Variables Types
numberVariable = 123
floatingPointNumber = 234.345
stringVariable = Hello World!
booleanVariable = true
isNumber = number
isString = string
isBoolean = boolean
```

2.2.3 Boolean Variables

To practice with Boolean data types, create a component called ***BooleanVariables*** and import it in the ***Lab3*** component. 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 + ""} <br/>***. Add ***function***, ***export***, and other keywords as needed.

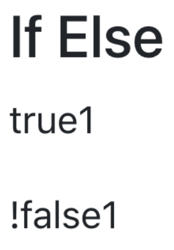
src/Labs/Lab3/BooleanVariables.tsx — *Browser*

```
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 id="wd-boolean-variables">
    <h4>Boolean Variables</h4>
    true1     = {true1 + ""}     <br />
    false1    = {false1 + ""}    <br />
    false2    = {false2 + ""}    <br />
    true2     = {true2 + ""}     <br />
    true3     = {true3 + ""}     <br />
    true4     = {true4 + ""}     <br />
    true5     = {true5 + ""}     <br />
    false3    = {false3 + ""}    <hr />
  </div>
);
```



```
Boolean Variables
true1 = true
false1 = false
false2 = false
true2 = true
true3 = true
true4 = true
true5 = true
false3 = false
```

2.2.4 Conditionals



```
If Else
true1
!false1
```

Conditional expressions allow scripts to make decisions based on predicates that compare values and variables. Scripts can decide to execute different parts of the code based on the result of these predicates using ***if/else*** and other constructs. Create the following components and import them into the ***Lab3*** component. Confirm that the components render as shown.

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/Lab3/IfElse.tsx

```
let true1 = true, false1 = false;
...
return(
   <div id="wd-if-else">
      <h4>If Else</h4>
      { true1 && <p>true1</p> }
      { !false1 ? <p>!false1</p> : <p>false1</p> } <hr/>
   </div>
)
```

2.2.5 Ternary Conditional Operator

Ternary conditional operators are concise alternative to ***if/else*** statements. It takes three arguments

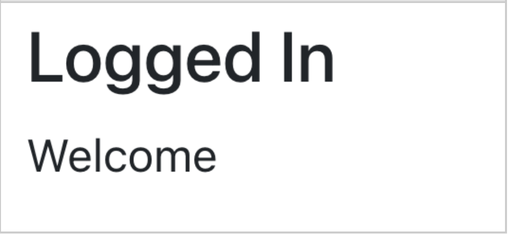
1. A ***predicate*** expression that evaluates to true or false followed by a question mark (?)
2. An expression that evaluates if the ***predicate*** is ***true*** followed by a colon (:)
3. Followed by an expression that evaluates iff the ***predicate*** is ***false***

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/Lab3/TernaryOperator.tsx

```
let loggedIn = true;
...
return(
   <div id="wd-ternary-operator">
      <h4>Logged In</h4>
      { loggedIn ? <p>Welcome</p> : <p>Please login</p> } <hr/>
   </div>
)
```

Browser



Logged In

Welcome

2.2.6 Generating conditional output

With boolean expressions we can render 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 following component to practice conditional rendering.

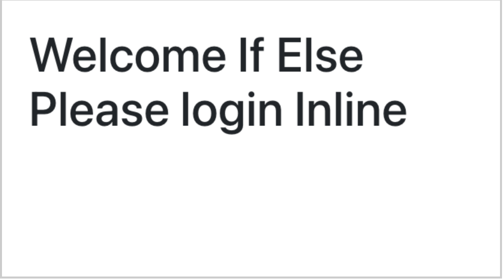
src/Labs/Lab3/ConditionalOutputIfElse.tsx

```
const ConditionalOutputIfElse = () => {
 const loggedIn = true;
 if(loggedIn) {
   return (<h2 id="wd-conditional-output-if-else-welcome">Welcome If Else</h2>);
 } else {
   return (<h2 id="wd-conditional-output-if-else-login">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 shown.

src/Labs/Lab3/ConditionalOutputInline.tsx	*Browser*

```
const ConditionalOutputInline = () => {
 const loggedIn = false;
 return (
   <div id="wd-conditional-output-inline">
     { loggedIn && <h2>Welcome Inline</h2>      }
     {!loggedIn && <h2>Please login Inline</h2> }
   </div>
 );
};
export default ConditionalOutputInline;
```



Welcome If Else
Please login Inline

2.3 JavaScript Functions

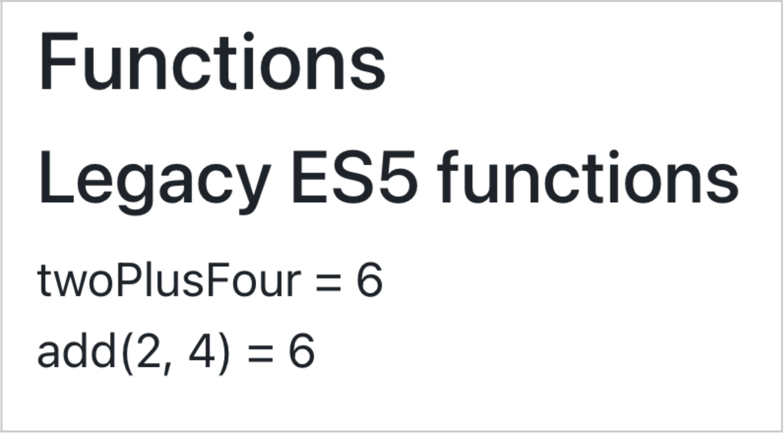
Functions allow reusing an algorithm by wrapping it in a named, parameterized block of code. ***JavaScript*** supports two styles of functions based on the history of language. Functions are declared using the following syntax.

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

To practice using functions create a new component called ***LegacyFunctions*** based on the code below. Import this new component in the ***Lab3*** component and confirm the browser renders as shown.

src/Labs/Lab3/LegacyFunctions.tsx	*Browser*

```
function add(a: number, b: number) {
  return a + b;
}
export default function LegacyFunctions() {
  const twoPlusFour = add(2, 4);
  console.log(twoPlusFour);
  return (
    <div id="wd-legacy-functions">
      <h4>Functions</h4>
      <h5>Legacy ES5 functions</h5>
      twoPlusFour = {twoPlusFour}
      <br />
      add(2, 4) = {add(2, 4)}
      <hr />
    </div>
);}
```



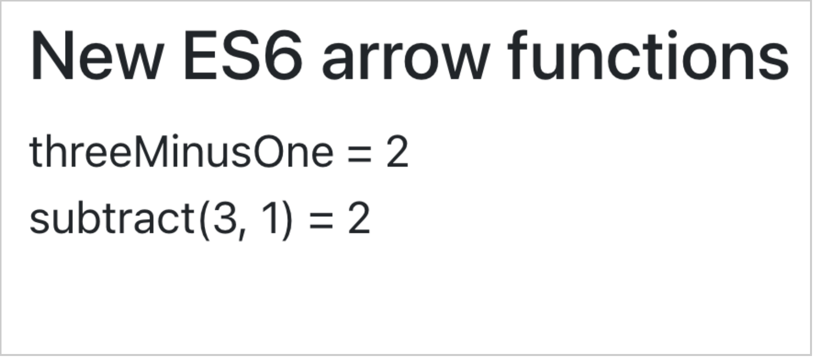
Functions
Legacy ES5 functions
twoPlusFour = 6
add(2, 4) = 6

2.3.1 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 the ***Lab3*** component and confirm the browser renders as shown.

src/Labs/Lab3/ArrowFunctions.tsx — ***Browser***

```
const subtract = (a: number, b: number) => {
  return a - b;
};
export default function ArrowFunctions() {
  const threeMinusOne = subtract(3, 1);
  console.log(threeMinusOne);
  return (
    <div id="wd-arrow-functions">
      <h4>New ES6 arrow functions</h4>
      threeMinusOne = {threeMinusOne}   <br />
      subtract(3, 1) = {subtract(3, 1)} <hr />
    </div>
);}
```



New ES6 arrow functions

threeMinusOne = 2

subtract(3, 1) = 2

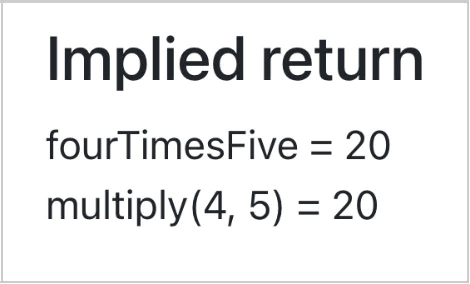
NOTE: Throughout the last couple of exercises we've provided code in the return statement to render the variables in the browser and asked that you confirm the output matches. Going forward we'll omit the return statement, but continue to implement it and confirm the output matches the one provided.

2.3.2 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 the ***Lab3*** component and confirm the browser renders as shown.

src/Labs/Lab3/ImpliedReturn.tsx — ***Browser***

```
const multiply = (a: number, b: number) => a * b;
const fourTimesFive = multiply(4, 5);
console.log(fourTimesFive);
return (
  <div id="wd-implied-return">
    <h4>Implied return</h4>
    fourTimesFive = {fourTimesFive}<br />
    multiply(4, 5) = {multiply(4, 5)} <hr />
  </div>
);
```



Implied return

fourTimesFive = 20

multiply(4, 5) = 20

2.3.3 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, symilar to 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. In your return statement, wrap the HTML output in a ***DIV*** whose ***ID*** is ***wd-template-literals***. Do not hard code the results ***5***, ***Welcome home alice***, etc. Instead use the values of variables ***result1***, ***result2***, etc., to render the the output shown above.

src/Labs/Lab3/TemplateLiterals.tsx

```
export default function TemplateLiterals() {
  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"}`;
  return (
    <div id="wd-template-literals">
      <h4>Template Literals</h4>
      result1 = {result1}     <br />
      result2 = {result2}     <br />
      greeting1 = {greeting1} <br />
      greeting2 = {greeting2} <hr />
    </div>
  );
}
```

Browser

```
Template Literals

result1 = 2 + 3 = 5
result2 = 2 + 3 = 5
greeting1 = Welcome home alice
greeting2 = Logged in: No
```

2.4 JavaScript Data Structures

Up to this point we have been discussing ***primitive datatypes*** such as ***strings***, ***numbers***, and ***booleans***. These can be combined into ***complex datatypes*** such as ***arrays*** and ***objects***. 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 actually do that. To practice with arrays create a component called ***SimpleArrays*** and use the code below as a guide to rendering the content on the right. Import the component to the ***Lab3*** component and confirm the browser renders as shown. Note that the arrays render without the commas. This feature will come in handy when the array items are ***HTML*** elements.

src/Labs/Lab3/SimpleArrays.tsx

```
export default function SimpleArrays() {
  var functionScoped = 2;
  let blockScoped = 5;
  const constant1 = functionScoped - blockScoped;
  let numberArray1 = [1, 2, 3, 4, 5];
  let stringArray1 = ["string1", "string2"];
  let htmlArray1 = [<li>Buy milk</li>, <li>Feed the pets</li>];
  let variableArray1 = [ functionScoped, blockScoped, constant1,
                         numberArray1, stringArray1 ];
  return (
    <div id="wd-simple-arrays">
      <h4>Simple Arrays</h4>
      numberArray1 = {numberArray1}     <br />
      stringArray1 = {stringArray1}     <br />
      variableArray1 = {variableArray1} <br />
      Todo list:
      <ol>{htmlArray1}</ol>
      <hr />
    </div>
  );
}
```

Browser

```
Simple Arrays

numberArray1 = 12345
stringArray1 = string1string2
variableArray1 = 25-312345string1string2
Todo list:
  1. Buy milk
  2. Feed the pets
```

2.4.1 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 ***Lab3*** and confirm the browser renders as shown.

src/Labs/Lab3/ArrayIndexAndLength.tsx

```
export default function ArrayIndexAndLength() {
  let numberArray1 = [1, 2, 3, 4, 5];
  const length1 = numberArray1.length;
  const index1 = numberArray1.indexOf(3);
  return (
    <div id="wd-array-index-and-length">
      <h4>Array index and length</h4>
      length1 = {length1} <br />
      index1 = {index1}   <hr />
    </div>
);}
```

Browser

```
Array index and length

length1 = 5
index1 = 2
```

2.4.2 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 ***AddingAndRemoving DataToFromArrays*** based on the code below. Import this new component in ***Lab3*** and confirm the browser renders as shown.

src/Labs/Lab3/AddingAndRemovingToFromArrays.tsx

```
export default function AddingAndRemovingToFromArrays() {
  let numberArray1 = [1, 2, 3, 4, 5];
  let stringArray1 = ["string1", "string2"];
  let todoArray = [<li>Buy milk</li>, <li>Feed the pets</li>];
  numberArray1.push(6); // adding new items
  stringArray1.push("string3");
  todoArray.push(<li>Walk the dogs</li>);
  numberArray1.splice(2, 1); // remove 1 item starting at 2
  stringArray1.splice(1, 1);
  return (
    <div id="wd-adding-removing-from-arrays">
      <h4>Add/remove to/from arrays</h4>
      numberArray1 = {numberArray1} <br />
      stringArray1 = {stringArray1} <br />
      Todo list:
      <ol>{todoArray}</ol><hr />
    </div>
);}
```

Browser

```
Add/remove to/from arrays

numberArray1 = 12456
stringArray1 = string1string3
Todo list:
  1. Buy milk
  2. Feed the pets
  3. Walk the dogs
```

2.4.3 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 ***Lab3*** and confirm the browser renders as shown.

src/Labs/Lab3/ForLoops.tsx

```
export default function ForLoops() {
  let stringArray1 = ["string1", "string3"];
  let stringArray2 = [];
  for (let i = 0; i < stringArray1.length; i++) {
    const string1 = stringArray1[i];
    stringArray2.push(string1.toUpperCase());
  }
  return (
    <div id="wd-for-loops">
      <h4>For Loops</h4>
      stringArray2 = {stringArray2} <hr />
    </div>
);}
```

Browser

```
Looping through arrays

stringArray2 = STRING1STRING3
```

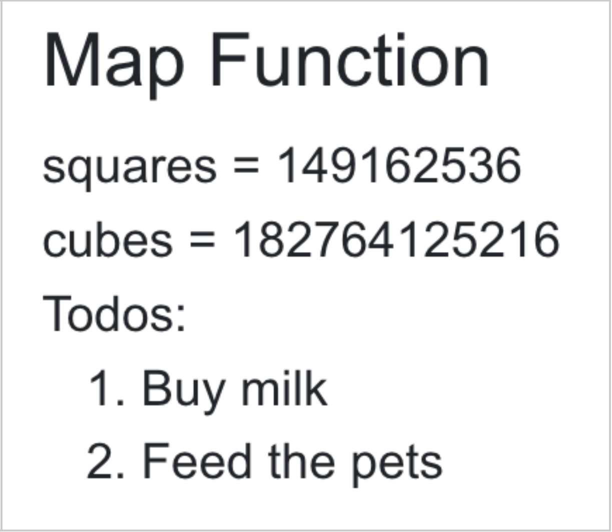
2.4.4 The Map Function

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/Lab3/MapFunction.tsx

```
export default function MapFunction() {
  let numberArray1 = [1, 2, 3, 4, 5, 6];
  const square = (a: number) => a * a;
  const todos = ["Buy milk", "Feed the pets"];
  const squares = numberArray1.map(square);
  const cubes = numberArray1.map((a) => a * a * a);
  return (
    <div id="wd-map-function">
      <h4>Map Function</h4>
      squares = {squares} <br />
      cubes = {cubes} <br />
      Todos:
      <ol>
        {todos.map((todo) => (
          <li>{todo}</li>
        ))}
      </ol> <hr/>
    </div>
  );
}
```

Browser



Map Function

squares = 149162536

cubes = 182764125216

Todos:

1. Buy milk
2. Feed the pets

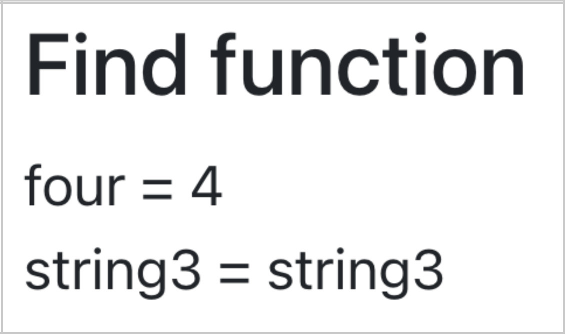
2.4.5 The Find Function

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 ***JavaScript*** and confirm the browser renders as shown.

src/Labs/Lab3/FindFunction.tsx

```
export default function FindFunction() {
  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");
  return (
    <div id="wd-find-function">
      <h4>Find Function</h4>
      four = {four} <br />
      string3 = {string3} <hr />
    </div>
);}
```

Browser



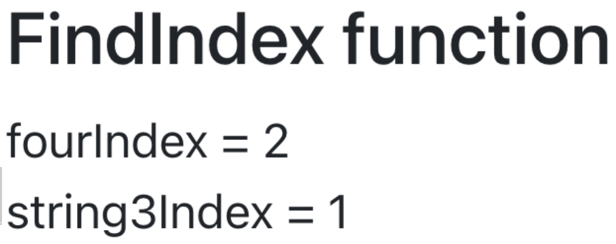
Find function

four = 4

string3 = string3

2.4.6 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.



FindIndex function

fourIndex = 2

string3Index = 1

src/Labs/Lab3/FindIndex.tsx

```
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');
```

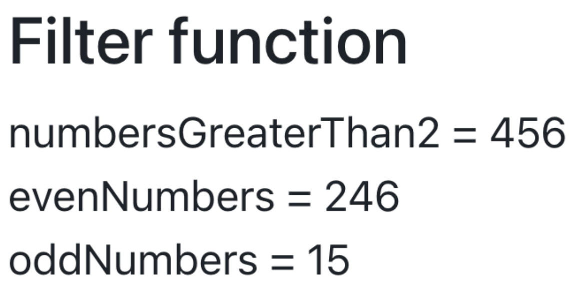
2.4.7 The Filter Function

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 ***Lab3*** and confirm the browser renders as shown.

src/Labs/Lab3/FilterFunction.tsx

```
export default function FilterFunction() {
  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);
  return (
    <div id="wd-filter-function">
      <h4>Filter Function</h4>
      numbersGreaterThan2 = {numbersGreaterThan2}  <br />
      evenNumbers = {evenNumbers}    <br />
      oddNumbers = {oddNumbers}      <hr />
    </div>
);}
```

Browser



Filter function

numbersGreaterThan2 = 456

evenNumbers = 246

oddNumbers = 15

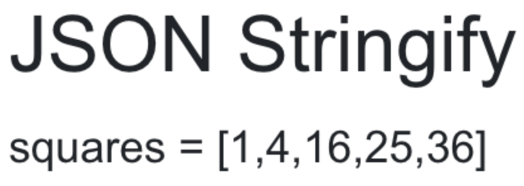
2.4.8 JSON Stringify

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*** data structures 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/Lab3/JsonStringify.tsx

```
export default function JsonStringify() {
  const squares = [1, 4, 16, 25, 36];
  return (
    <div className="wd-json-stringify">
      <h3>JSON Stringify</h3>
      squares = {JSON.stringify(squares)}
      <hr />
    </div>
  );
}
```

Browser



JSON Stringify

squares = [1,4,16,25,36]

2.4.9 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 ***Lab3*** component, and confirm it renders as shown below.

src/Labs/Lab3/House.tsx

```
export default 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 id="wd-house">
      <h4>House</h4>
      <h5>bedrooms</h5>
      {house.bedrooms}
      <h5>bathrooms</h5>
      {house.bathrooms}
      <h5>Data</h5>
      <pre>{JSON.stringify(house, null, 2)}</pre>
      <hr />
    </div>
  );
}
```

Browser

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"
  ]
}
```

2.4.10 Rendering a Data Structure

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/Lab3/todo***, implement the ***TodoItem*** component in a ***TodoItem.tsx*** file as shown below. Import the component into the ***Lab3*** component and confirm that it renders as shown.

src/Labs/Lab3/todos/TodoItem.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;
```

Browser

☑ Buy milk(COMPLETED)

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

src/Labs/Lab3/todos/todos.json

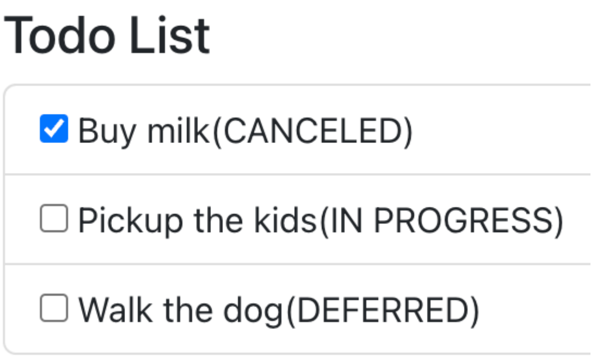
```
[
  { "title": "Buy milk",          "status": "CANCELED",     "done": true   },
  { "title": "Pickup the kids",   "status": "IN PROGRESS",  "done": false  },
  { "title": "Walk the dog",      "status": "DEFERRED",     "done": false  }
]
```

Now let's implement a ***TodoList*** component that renders the array of todos as shown below. Import the component in ***Labs/Lab3/index.tsx***, refresh the browser, and confirm the ***TodoList*** renders a list of checkboxes and todo items.

/src/Labs/Lab3/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><hr/>
   </>
 );
}
export default TodoList;
```

Browser



Todo List

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

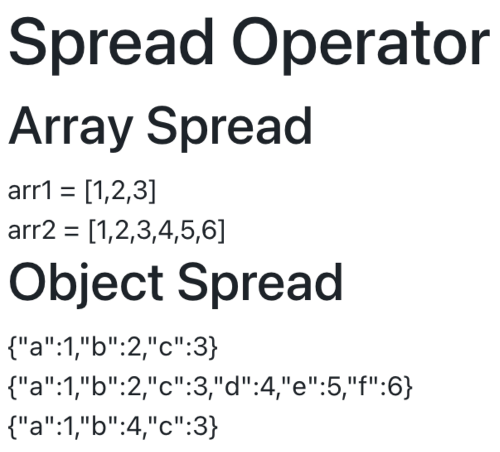
2.4.11 The Spread Operator

The spread operator (...) is used to expand, or copy an iterable object or array into another object or array. In the example below we declare array ***arr1*** and then copy its content (spread) into array ***arr2***. The resulting array ***arr2*** contains the contents of ***arr1***, followed by the rest of the items already 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*** onto ***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 a 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 ***Spreading*** component as shown below, import it in the ***Lab3*** component and confirm it renders as shown below.

src/Labs/Lab3/Spreading.tsx

```
export default 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 id="wd-spreading">
     <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 />  <hr />
   </div>);
}
```

Browser



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}

2.4.12 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 ***Lab3*** component, and confirm it renders as shown below.

src/Labs/Lab3/Destructing.tsx

```
export default function Destructing() {
 const person = { name: "John", age: 25 };
 const { name, age } = person;
 // const name = person.name
 // const age = person.age
 const numbers = ["one", "two", "three"];
 const [ first, second, third ] = numbers;
 return (
   <div id="wd-destructing">
     <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}<hr />
   </div>
 );
}
```

Browser

```
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
```

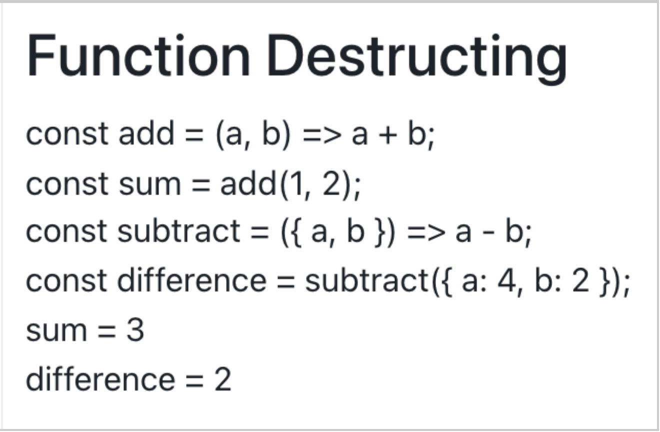
2.4.13 Destructing Function Parameters

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 ***Lab3*** component, and confirm it renders as shown below on the right.

src/Labs/Lab3/FunctionDestructing.tsx

```
export default 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 id="wd-function-destructing">
     <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} <hr />
   </div>
);}
```

Browser



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

2.4.14 Destructing Imports

Let's create a simple library to illustrate various ways of importing the functions and constants declared in the ***Math*** library below. The functions ***add***, ***subtract***, ***multiply***, and ***divide*** are all exported with the ***export*** keyword so that they can be imported individually. The ***Math*** constant declares an object containing references to the local functions. We export the ***Math*** object as the ***default export*** so that the functions can be imported as a single object map.

src/Labs/Lab3/Math.ts

```
export function add(a: number, b: number): number {
  return a + b;
}
export function subtract(a: number, b: number): number {
  return a - b;
}
export function multiply(a: number, b: number): number {
  return a * b;
}
export function divide(a: number, b: number): number {
  return a / b;
}
const Math = {
  add,
  subtract,
  multiply,
  divide,
};
export default Math;
```

To demonstrate how the functions can be imported in several ways, create the ***DestructingImports*** component below. The component implements a table we're going to fill out with three different ways we can import the functions from the ***Math*** library we implemented above. First, the functions can be imported as the single ***Math*** object that contains references to all the functions as ***import Math from "./Math"***. Then we can access each of the functions through the ***Math*** instance as ***Math.add()***, ***Math.subtract()***, etc. An alternative way to import the functions as a single object is using the ***import * as NAME from "./Math"***, where ***NAME*** is a custom local object name, e.g., ***Matematica***. We can then using the object's name to invoke the functions as ***Matematica.add()***, ***Matematica.subtract()***, etc. Finally, the functions can be imported individually by destructing the exported functions as ***import {add, subtract, multiply, divide} from "./Math"***.

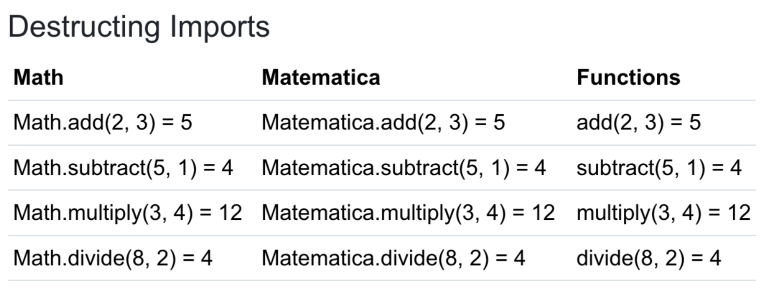
src/Labs/Lab3/DestructingImports.tsx

```
import React from "react";
import Math, { add, subtract, multiply, divide } from "./Math";
import * as Matematica from "./Math";
export default function DestructingImports() {
  return (
    <div id="wd-destructuring-imports">
      <h2>Destructing Imports</h2>
      <table className="table table-sm">
        <thead>
          <tr>
            <th>Math</th>
            <th>Matematica</th>
            <th>Functions</th>
          </tr>
        </thead>
        <tbody>
          {/* see next code block */}
        </tbody>
      </table>
      <hr />
    </div>
);}
```

Here are several examples demonstrating using the functions through the objects ***Math***, ***Matematica***, and then using the functions individually. Implement the ***Math.ts*** library and the ***DestructingImports*** component and confirm it renders as shown. Complete missing code.

src/Labs/Lab3/DestructingImports.tsx

```
<tr>
  <td>Math.add(2, 3) = {Math.add(2, 3)}</td>
  <td>Matematica.add(2, 3) =
     {Matematica.add(2, 3)}</td>
  <td>add(2, 3) = {add(2, 3)}</td>
</tr>
<tr>
  <td>Math.subtract(5, 1) = {Math.subtract(5, 1)}</td>
  <td>Matematica.subtract(5, 1) =
     {Matematica.subtract(5, 1)}</td>
  <td>subtract(5, 1) = {subtract(5, 1)}</td>
</tr>
```

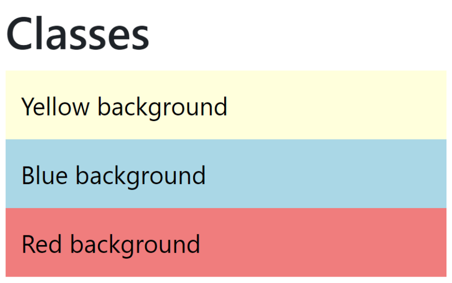


Destructing Imports

Math	Matematica	Functions
Math.add(2, 3) = 5	Matematica.add(2, 3) = 5	add(2, 3) = 5
Math.subtract(5, 1) = 4	Matematica.subtract(5, 1) = 4	subtract(5, 1) = 4
Math.multiply(3, 4) = 12	Matematica.multiply(3, 4) = 12	multiply(3, 4) = 12
Math.divide(8, 2) = 4	Matematica.divide(8, 2) = 4	divide(8, 2) = 4

2.5 Dynamic Styling

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.



Classes

Yellow background

Blue background

Red background

2.5.1 Working with HTML classes

Let's start practicing simple things, like classes and styles. Under the ***Labs/Lab3*** folder, create a new component ***Classes*** with a matching styling file.

src/Labs/Lab3/Classes.tsx

```
import './Classes.css';
export default function Classes() {
  return (
    <div>
```

src/Labs/Lab3/Classes.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;  }
```

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

```
.wd-fg-black     { color: black;                         }
.wd-padding-10px { padding: 10px;                        }
```

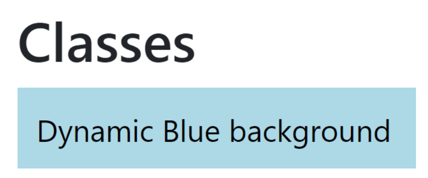
From the ***Lab3*** component, import the new ***Classes*** component and confirm the component renders as shown.

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

src/Labs/Lab3/Classes.tsx

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

Browser



Classes

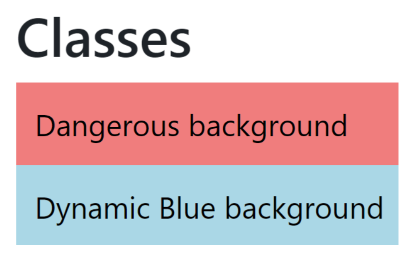
Dynamic Blue background

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.

src/Labs/Lab3/Classes.tsx

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

Browser



Classes

Dangerous background

Dynamic Blue background

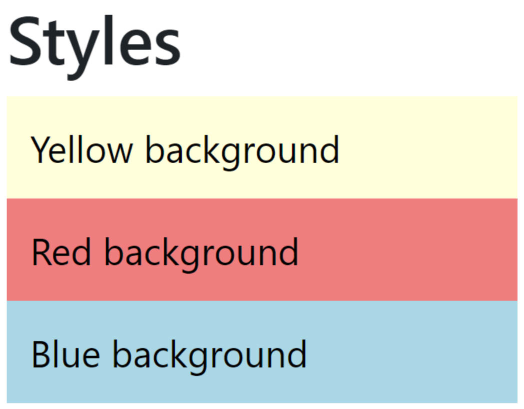
2.5.2 Working with the HTML Style attribute

In React.js, the ***styles*** attribute 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 and then import it into the ***Lab3*** component. The ***Styles*** component declares constant JSON objects that can be applied to elements using the ***styles*** attribute. Alternatively, the styles attribute accepts a JSON literal object instance which results in a weird syntax of double curly brackets. Refresh the browser and confirm the browser renders as expected.

Labs/Lab3/Styles.tsx

```
export default function Styles() {
  const colorBlack = { color: "black" };
  const padding10px = { padding: "10px" };
  const bgBlue = { "backgroundColor": "lightblue",
                   "color": "black",   ...padding10px
  };
  const bgRed = {
    "backgroundColor": "lightcoral",
    ...colorBlack,   ...padding10px
  };
  return(
    <div id="wd-styles">
      <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>
);};
```

Browser



Styles

Yellow background

Red background

Blue background

2.6 Parameterizing Components

React components can be parameterized by using the familiar HTML attribute syntax, which passes attribute values to the component's function as an object map parameter. The following ***Add*** component can receive properties ***a*** and ***b*** deconstructed from the attributes of its equivalent HTML attributes syntax. Implement the ***Add*** component below and confirm that passing it ***a=3*** and ***b=4*** results in ***a + b = 7***. Note that the values of ***a*** and ***b*** are descructed from the object parameter in the ***Add*** function parameter list.

src/Labs/Lab3/Add.tsx

```
export default function Add({ a, b }: { a: number; b: number }) {
  return (
    <div id="wd-add">
      <h4>Add</h4>a = {a}
      b = {b}         <br />
      a + b = {a + b} <hr />
    </div>
  );
}
```

src/Labs/Lab3/index.tsx

```
import Add from "./Add";
export default function Lab3() {
  return (
    <div id="wd-lab3" className="container">
      <h3>Lab 3</h3>
      ...
      <Add a={3} b={4} />
    </div>
  );
}
```

2.6.1 Child Components

In the previous section we discussed passing data to a component through attributes. Another way to pass data to a component is in its body, that is, between the opening and closing tag of the element. In HTML it's common to wrap content with specific tags to add certain formatting. For instance the tags ***h1*** and ***p*** shown below format the content in their bodies with specific font sizes and margins. Basically these elements take the content in the body and return a transformed version of the content.

```
<h1>This content is formatted as a heading of size 1</h1>
<p>This content is formatted as a paragraph</p>
```

We can implement ***React*** components to take content in their body and return a new version of that content. The content in the body of a ***React*** component is passed to the component function as parameter called ***children***. For instance, the component below takes a number in its body and returns the square of the number. Import the new component, use to compute the square of 4 and confirm it renders the correct result *src/Labs/Lab3/Square.tsx*

```
import React, { ReactNode } from "react";
export default function Square({ children }: { children: ReactNode }) {
  const num = Number(children);
  return <span id="wd-square">{num * num}</span>;
}
```

src/Labs/Lab3/index.tsx

```
import Square from "./Square";
export default function Lab3() {
  return (
    <div id="wd-lab3">
      <h3>JavaScript</h3>
      ...
      <h4>Square of 4</h4>
      <Square>4</Square>
      <hr />
    </div>
  );
}
```

Here's another example that highlights the content in its body making the content red on yellow. The example below receives the ***children*** property as a property and then renders 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.

src/Labs/Lab3/Highlight.tsx

```
import { ReactNode } from "react";
export default function Highlight({ children }: { children: ReactNode }) {
  return (
    <span id="wd-highlight" style={{ backgroundColor: "yellow", color: "red" }}>
      {children}
    </span>
  );
}
```

Implement the ***Highlight*** component as shown above, import it in ***Lab3***, and confirm it highlights the content shown.

src/Labs/Lab3/index.tsx

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

export default function Lab3() {
 return (
   <div id="wd-lab3">
     <h3>Lab 3</h3>
     ...
     <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>
 );}
```

2.6.2 Working with Location

The ***useLocation()*** hook returns several properties related to the current URL, in particular the ***pathname*** property containing 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 ***Lab1***, ***Lab2***, or ***Lab3*** to then add the ***active*** class to highlight the correct pill. Confirm that the correct tab highlights when you click each of the pills.

src/Labs/TOC.tsx

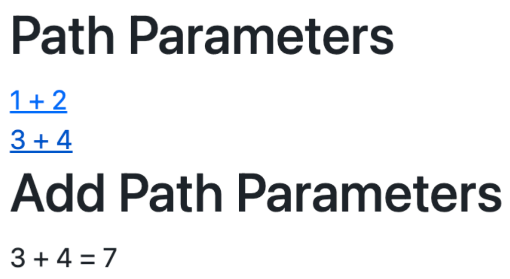
```
import { useLocation } from "react-router";
export default function TOC() {
  const { pathname } = useLocation();
  return (
    <ul className="nav nav-pills" id="wd-toc">
      <li className="nav-item"><a id="wd-a"  href="#/Labs" className="nav-link">Labs</a></li>
      <li className="nav-item"><a id="wd-a1" href="#/Labs/Lab1"
          className={`nav-link ${pathname.includes("Lab1") ? "active" : ""}`}>Lab 1</a></li>
      <li className="nav-item"><a id="wd-a2" href="#/Labs/Lab2"
          className={`nav-link ${pathname.includes("Lab2") ? "active" : ""}`}>Lab 2</a></li>
      <li className="nav-item"><a id="wd-a3" href="#/Labs/Lab3"
          className={`nav-link ${pathname.includes("Lab3") ? "active" : ""}`}>Lab 3</a></li>
      <li className="nav-item"><a id="wd-k" href="#/Kanbas" className="nav-link">Kanbas</a></li>
      <li className="nav-item"><a id="wd-github" href="https://github.com/jannunzi" target="_blank"
          className="nav-link">My GitHub</a></li>
    </ul>
);}
```

2.6.3 Encoding Path Parameters

The URL can also be used to encode parameters when navigating between screens. Components 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.

src/Labs/Lab3/AddPathParameters.tsx

```
import React from "react";
import { useParams } from "react-router-dom";
export default function AddPathParameters() {
  const { a, b } = useParams();
  return (
    <div id="wd-add"> <h4>Add Path Parameters</h4>
      {a} + {b} = {parseInt(a as string) + parseInt(b as string)}
    </div>
  );
}
```


Path Parameters

1 + 2
3 + 4

Add Path Parameters

3 + 4 = 7

Path parameter names such as ***a*** and ***b***, are declared in the ***path*** attribute of the ***Route*** component for the target screen. 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 ***Lab3*** 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***. Add '***/****' to the ***Lab3*** route in the ***Labs*** component as shown below on the right.

src/Labs/Lab3/PathParameters.tsx

```
import { Routes, Route, Link } from "react-router-dom";
import AddPathParameters from "./AddPathParameters";
export default function PathParameters() {
  return (
    <div id="wd-path-parameters">
      <h2>Path Parameters</h2>
      <Link to="/Labs/Lab3/add/1/2">1 + 2</Link> <br />
      <Link to="/Labs/Lab3/add/3/4">3 + 4</Link>
      <Routes>
        <Route path="add/:a/:b"
               element={<AddPathParameters />} />
      </Routes>
    </div>
  );
}
```

src/Labs/index.tsx

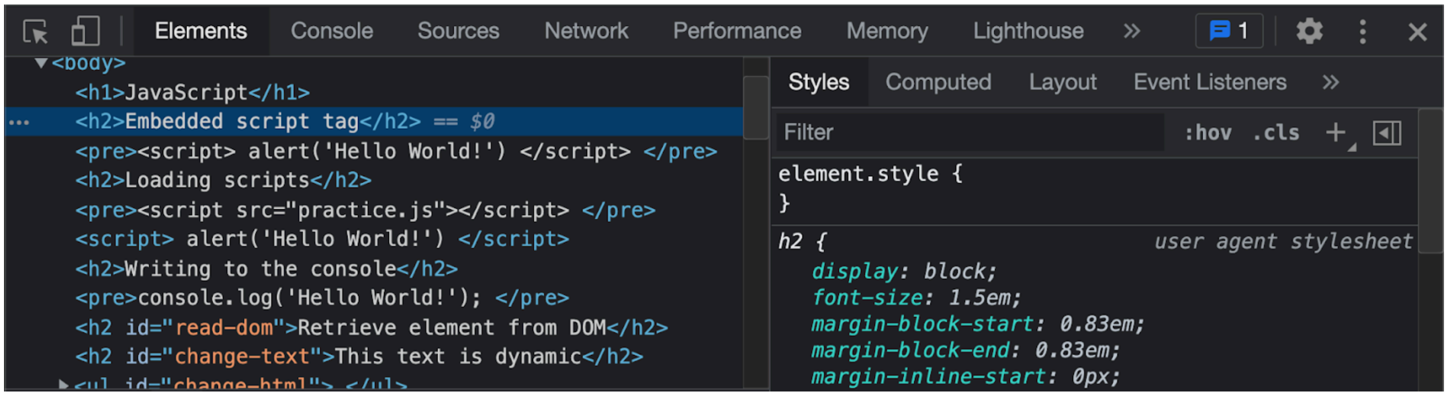
```
export default function Labs() {
  return (
    <div className="p-3">
      <h1>Labs</h1>
      <TOC />
      <Routes>
        <Route path="Lab1" element={<Lab1 />} />
        <Route path="Lab2" element={<Lab2 />} />
        <Route path="Lab3/*" element={<Lab3 />} />
      </Routes>
    </div>
  );
}
```

2.7 Debugging

The ***Web Dev Tools*** provide several tools to analyze various types of source code common in the Web development process. So far we've looked at the ***Elements*** and ***Styles*** tab where we can analyze the ***DOM*** data structure generated by the browser when it parsed the ***HTML*** or content dynamically generated by ***JavaScript***. Now that we have been implementing functions and algorithms, we should become familiar with the ***Console*** and ***Source*** tabs.

2.7.1 Writing to the Console from JavaScript

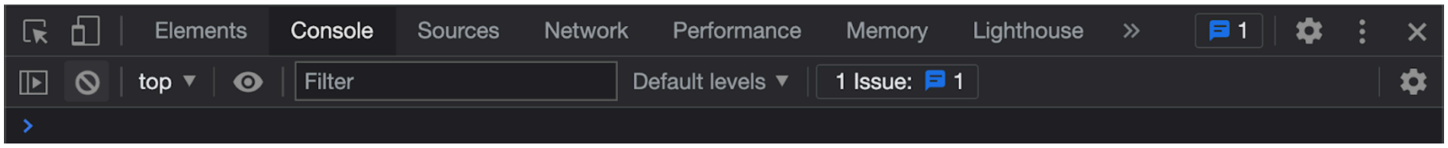
A useful feature of modern browsers is providing a development environment where developers can analyze the performance of their scripts. One way to analyze scripts are behaving correctly is to write output to the ***console*** from within scripts. To practice writing to the ***console***, bring up the 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.



```
▼<body>
    <h1>JavaScript</h1>
    <h2>Embedded script tag</h2> == $0
    <pre><script> alert('Hello World!') </script> </pre>
    <h2>Loading scripts</h2>
    <pre><script src="practice.js"></script> </pre>
    <script> alert('Hello World!') </script>
    <h2>Writing to the console</h2>
    <pre>console.log('Hello World!'); </pre>
    <h2 id="read-dom">Retrieve element from DOM</h2>
    <h2 id="change-text">This text is dynamic</h2>
```

```
element.style {
}
h2 {                                    user agent stylesheet
    display: block;
    font-size: 1.5em;
    margin-block-start: 0.83em;
    margin-block-end: 0.83em;
    margin-inline-start: 0px;
```

Click on the ***Console*** tab where we will be logging to throughout this assignment.



Add a ***console.log()*** statement to the ***Lab3*** component, reload the screen and confirm that "***Hello World!***" is displayed in the console.

src/Labs/Lab3/index.tsx	*Console*

```
export default function *Lab3*() {
   *console*.log('Hello World!');
   return(
      <div>
         <h3>JavaScript</h3>
         ...
      </div>
);}
```
	Hello World!

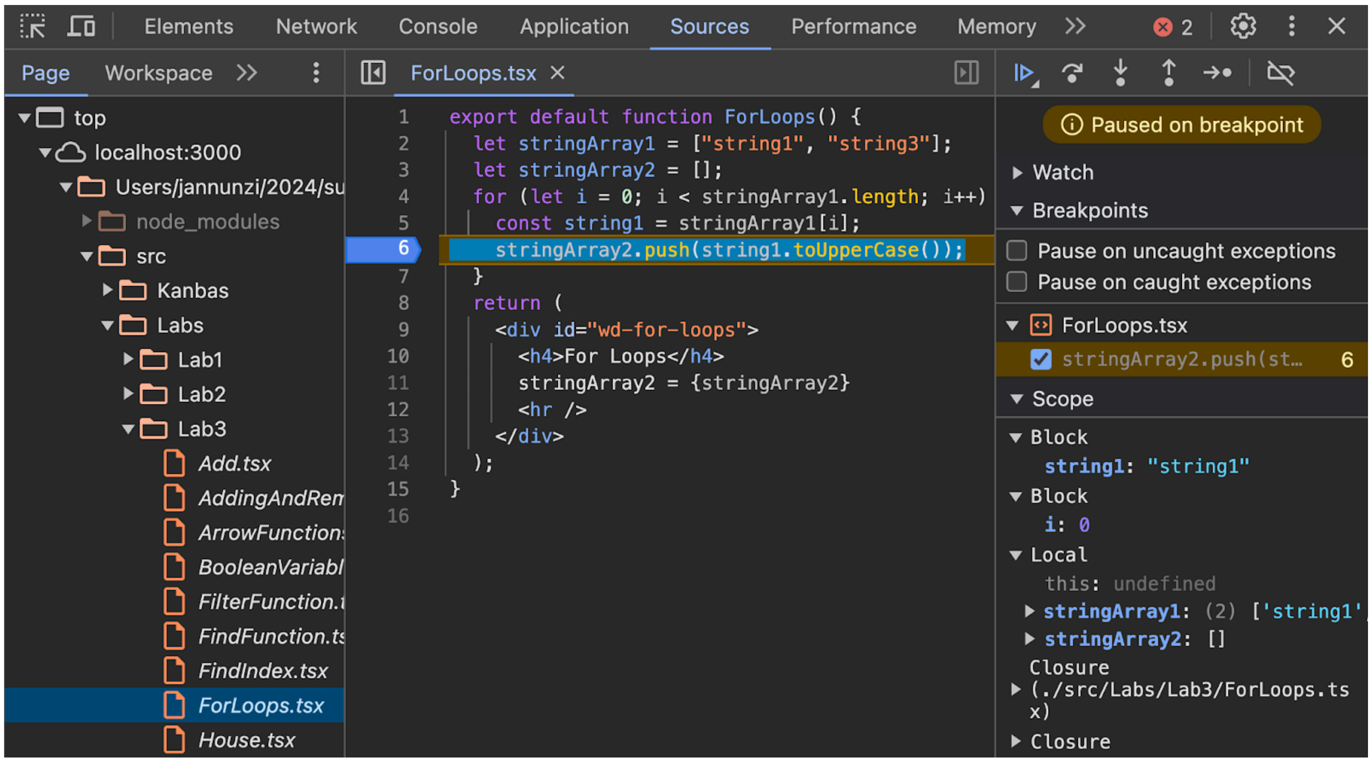
We can also display ***JSON*** data to confirm it is what we expect. Add a ***console.log()*** statement to print the ***house*** JSON object in the ***House*** component as shown below. Confirm the ***house*** object prints to the console.

src/Labs/Lab3/House.tsx	*Console*

```
export default function House() {
  const house = {
    ...
  };
  console.log(house);
  return (
    <div id="wd-house">
      ...
    </div>
  );
}
```
	```
	▼ {bedrooms: 4, bathrooms: 2.5, squareFeet: 2000, address: {…}, owners: Array(2)} i
	  ▼ address:
	      city: "Roma"
	      country: "Italy"
	      state: "RM"
	      street: "Via Roma"
	      zip: "00100"
	    ▶ [[Prototype]]: Object
	    bathrooms: 2.5
	    bedrooms: 4
	  ▶ owners: (2) ['Alice', 'Bob']
	    squareFeet: 2000
	```

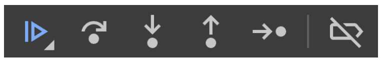
2.7.2 Breakpoints

Often analyzing code requires a more indepth effort of stepping through the code as its running. The ***Sources*** tab in the ***Web Dev Tools*** gives access to all the ***JavaScript*** code loaded by the browser so that developers can add breakpoints and step through the code. To practice debugging, open the ***Web Dev Tools*** and select the ***Sources*** tab as shown below. On the left sidebar, navigate through the folders looking for the ***ForLoops*** component we wrote earlier, and click on it to show its source code on the middle pane as shown below. Explore setting breakpoints on the line numbers and reload the page by pressing ***Ctrl+R*** on Windows or ***⌘+R*** on macOS. Confirm that the execution pauses on the breakpoint. In the example screenshot below we have added a breakpoint on line 6 of the ***ForLoops*** component and reloaded the page. Execution paused and line 6 is highlighted giving us developers an opportunity to analyze the current state of the application. On the right sidebar we can see the values of variables within the current scope such as ***string1***, ***stringArray1***, and ***stringArray2***.



```
1   export default function ForLoops() {
2     let stringArray1 = ["string1", "string3"];
3     let stringArray2 = [];
4     for (let i = 0; i < stringArray1.length; i++) {
5       const string1 = stringArray1[i];
6       stringArray2.push(string1.toUpperCase());
7     }
8     return (
9       <div id="wd-for-loops">
10        <h4>For Loops</h4>
11        stringArray2 = {stringArray2}
12        <hr />
13      </div>
14    );
15  }
16
```

```
☑ stringArray2.push(st…   6

▼ Block
    string1: "string1"
▼ Block
    i: 0
▼ Local
    this: undefined
  ▶ stringArray1: (2) ['string1'
  ▶ stringArray2: []
  Closure
▶ (./src/Labs/Lab3/ForLoops.ts
  x)
▶ Closure
```



The right sidebar provides buttons at the top to control the execution shown here on the right. The first button is for pausing or resuming execution. The second button is for stepping over the next function. The second button is for stepping into a function. The third button is for stepping out of the current function. The fifth button is for stepping one line at a time. The last button is for enabling and disabling breakpoints.

3 Implementing a Data Driven Kanbas Application

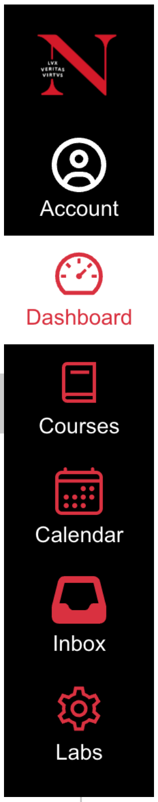
In previous assignments implemented the ***Kanbas*** application as shown below. The current implementation uses ***React*** components to delegate the creation of various components and screens that are then aggregated into a single application. This assignment will revisit the implementation by making it data driven. Instead of displaying a single course, a single set of modules and assignments, ***JSON*** data files will drive the rendering as users navigate from the ***Dashboard*** to a particular course.

src/Kanbas/index.tsx

```
export default function Kanbas() {
  return (
    <div id="wd-kanbas">
      <KanbasNavigation />
      <div className="wd-main-content-offset p-3">
        <Routes>
          <Route path="/" element={<Navigate to="Account" />} />
          <Route path="/Account/*" element={<Account />} />
          <Route path="/Dashboard" element={<Dashboard />} />
          <Route path="/Courses/:cid/*" element={<Courses />} />
          <Route path="/Calendar" element={<h1>Calendar</h1>} />
          <Route path="/Inbox" element={<h1>Inbox</h1>} />
        </Routes>
      </div>
    </div>
);}
```

3.1 Data Driven Kanbas Navigation

The current implementation of the ***KanbasNavigation*** component consists of a hardcoded list of links to navigate to the ***Dashboard*** and other places in ***Kanbas***. Instead of hardcoding the list in HTML, create a data structure that configures the ***label***s, ***path***s, and ***icon***s as an array and then map over the data structure creating the links dynamically. Here's an example of how to implement the ***Kanbas Navigation*** component. Confirm the sidebar renders as shown here on the right and the navigation still works. Note that the ***Courses*** link has been intentionally configured to navigate to the ***Dashboard***, since it only makes sense to navigate to the ***Courses*** screen from the ***Dashboard***.



src/Kanbas/Navigation.tsx

```
import { AiOutlineDashboard } from "react-icons/ai";
import { IoCalendarOutline } from "react-icons/io5";
import { LiaBookSolid, LiaCogSolid } from "react-icons/lia";
import { FaInbox, FaRegCircleUser } from "react-icons/fa6";
import { Link, useLocation } from "react-router-dom";
export default function KanbasNavigation() {
  const { pathname } = useLocation();
  const links = [
    { label: "Dashboard", path: "/Kanbas/Dashboard", icon: AiOutlineDashboard },
    { label: "Courses",   path: "/Kanbas/Dashboard", icon: LiaBookSolid },
    { label: "Calendar",  path: "/Kanbas/Calendar",  icon: IoCalendarOutline },
    { label: "Inbox",     path: "/Kanbas/Inbox",     icon: FaInbox },
    { label: "Labs",      path: "/Labs",             icon: LiaCogSolid },
  ];
  return (
    <div id="wd-kanbas-navigation" style={{width: 120}}
         className="list-group rounded-0 position-fixed bottom-0 top-0 d-none d-md-block bg-black z-2">
      <a id="wd-neu-link" target="_blank" href="https://www.northeastern.edu/"
        className="list-group-item bg-black border-0 text-center">
        <img src="/images/NEU.png" width="75px" /></a>
      <Link to="/Kanbas/Account" className={`list-group-item text-center border-0 bg-black
            ${pathname.includes("Account") ? "bg-white text-danger" : "bg-black text-white"}`}>
        <FaRegCircleUser className={`fs-1 ${pathname.includes("Account") ? "text-danger" : "text-white"}`} />
        <br />
        Account
      </Link>
      {links.map((link) => (
        <Link key={link.path} to={link.path} className={`list-group-item bg-black text-center border-0
              ${pathname.includes(link.label) ? "text-danger bg-white" : "text-white bg-black"}`}>
          {link.icon({ className: "fs-1 text-danger"})}
          <br />
          {link.label}
        </Link>
      ))}
    </div>
);}
```

3.2 Implementing a Kanbas "Database"

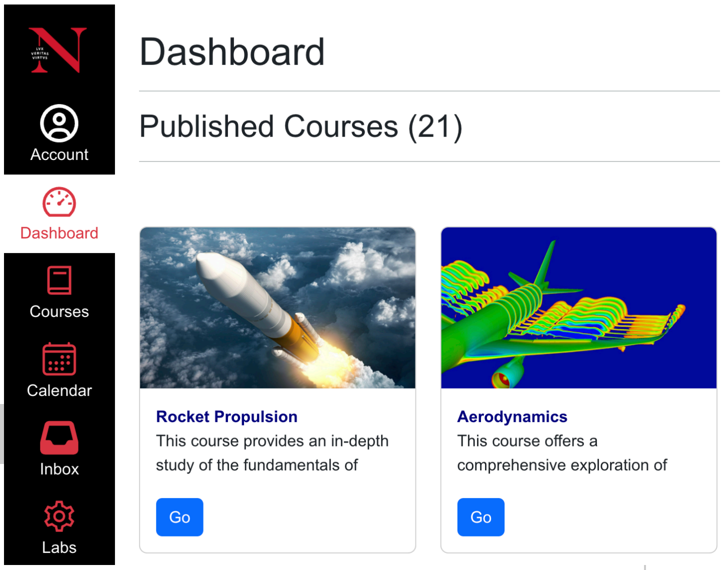
Implement the data file that will contain all the data needed in the ***Kanbas*** application. We'll start by adding courses in a ***courses.json*** file under a new ***Database*** folder. Download a sample data file containing the courses from my gist on GitHub. Save the file to ***src/Kanbas/Database/courses.json***. 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.ts

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

3.3 Data Driven Dashboard Screen

Based on your implementation of the ***Dashboard*** in previous assignments, refactor the component so that it dynamically renders the ***courses*** in the ***Database***. The following code is an example of how you can implement the ***Dashboard*** component dynamically mapping through an array of ***courses***. Confirm that the ***Dashboard*** component renders the courses as a grid and that clicking on a course navigates to the ***Home*** screen, but now the ***URL*** encodes the ***ID*** of the course.



Dashboard

Published Courses (21)

Rocket Propulsion
This course provides an in-depth study of the fundamentals of
Go

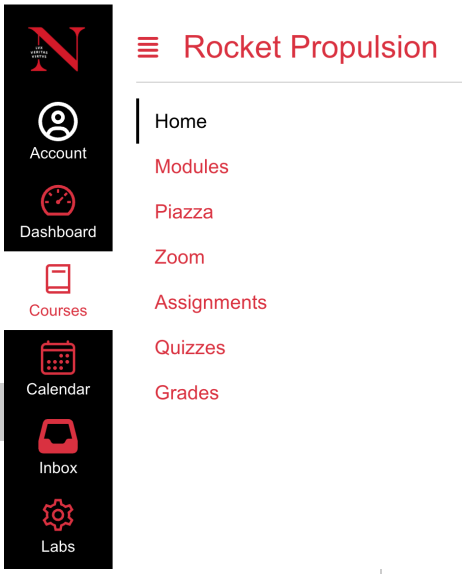
Aerodynamics
This course offers a comprehensive exploration of
Go

src/Kanbas/Dashboard.tsx

```
import { Link } from "react-router-dom";
import * as db from "./Database";
export default function Dashboard() {
  const courses = db.courses;
  return (
    <div id="wd-dashboard">
      <h1 id="wd-dashboard-title">Dashboard</h1> <hr />
      <h2 id="wd-dashboard-published">Published Courses ({courses.length})</h2> <hr />
      <div id="wd-dashboard-courses" className="row">
        <div className="row row-cols-1 row-cols-md-5 g-4">
          {courses.map((course) => (
            <div className="wd-dashboard-course col" style={{ width: "300px" }}>
              <div className="card rounded-3 overflow-hidden">
                <Link to={`/Kanbas/Courses/${course._id}/Home`}
                      className="wd-dashboard-course-link text-decoration-none text-dark" >
                  <img src="/images/reactjs.jpg" width="100%" height={160} />
                  <div className="card-body">
                    <h5 className="wd-dashboard-course-title card-title">
                      {course.name}
                    </h5>
                    <p className="wd-dashboard-course-title card-text overflow-y-hidden" style={{ maxHeight: 100 }}>
                      {course.description}
                    </p>
                    <button className="btn btn-primary"> Go </button>
                  </div>
                </Link>
              </div>
            </div>
          ))}
        </div>
      </div>
    </div>
);}
```

The ***Link*** component is used to create the hyperlinks and encode the course's ID as part of the path. This can then be used in other components to parse the ID from the path and display content specific to the selected course. Confirm that clicking on different courses encodes the corresponding course ID in the path. Refactor the ***courses.json*** and the ***Dashboard*** component so that each course has a different image.

3.4 Data Driven Course Screen



Now that clicking on a course on the ***Dashboard*** encodes the course's ***ID*** on the path, we can refactor the ***Courses*** screen to parse the course's ***ID*** from the path and then render the corresponding course's name. The ***Route*** for the ***Courses*** component encodes the course's ***id*** in the ***path*** attribute and is available to the ***Courses*** component through the ***useParams()*** hook. Once we have the ***cid*** parameter, we can look up for the ***course*** from the ***Database*** for the ***course*** with the same ***_id***. Refactor the ***Courses*** component to render the ***name*** of the course you click on in the ***Dashboard*** as shown here on the right.

src/Kanbas/Courses/index.tsx

```
import { courses } from "../Database";
import { FaAlignJustify } from "react-icons/fa6";
import { Navigate, Route, Routes, useParams } from "react-router";
export default function Courses() {
  const { cid } = useParams();
  const course = courses.find((course) => course._id === cid);
  return (
    <div id="wd-courses">
      <h2 className="text-danger">
        <FaAlignJustify className="me-4 fs-4 mb-1" />
        {course && course.name}
      </h2>
      ...
    </div>
  );
}
```

3.5 Data Driven Course Navigation (On Your Own)

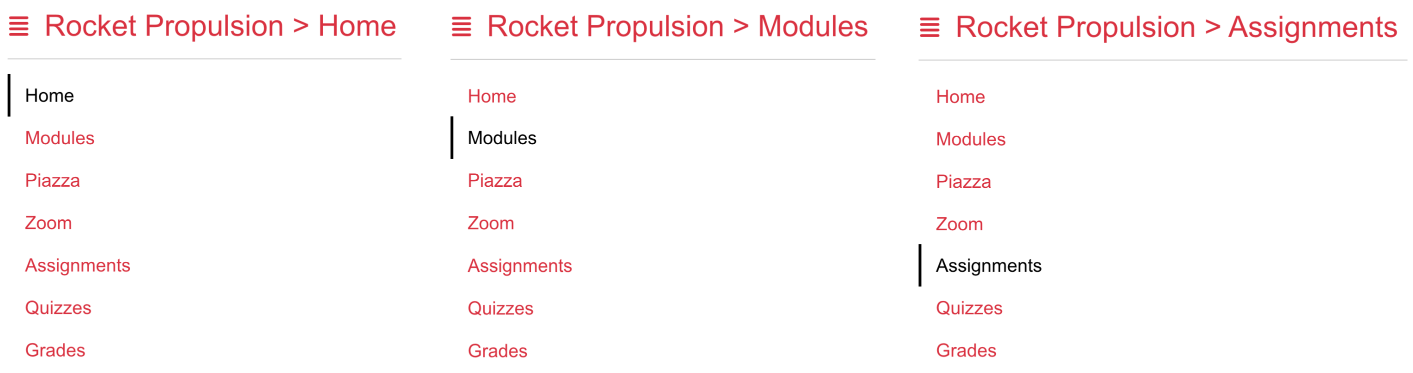
Based on the earlier ***Data Driven Kanbas Navigation*** example, refactor the ***Course Navigation*** sidebar so that it is not a list of hardcoded links and instead uses the following array.

```
  const links = ["Home", "Modules", "Piazza", "Zoom", "Assignments", "Quizzes", "Grades", "People"];
```

Use ***useParams()*** to retrieve the current course's ***ID*** and use ***useLocation()*** to retrieve the current ***pathname***. The links should highlight when you click on them and the URL should encode the course's ***ID*** in the path. Navigation to the ***Home***, ***Modules***, ***Assignments***, and ***Grades*** screen should still work as before.

3.6 Implementing the Breadcrumb Component

A ***breadcrumb*** is graphical representation that helps users know where they are within a set of screens. For instance the course name at the top of the ***Courses*** screen lets users know they are in a particular course. As they click through the links in the ***Courses Navigation*** sidebar, we can append the name of the section to the name of the course to further indicate what section are we in. The screenshots below illustrate navigating to the ***Home***, ***Modules***, and ***Assignments*** screens. In the ***Courses*** component, implement the breadcrumbs as shown below.



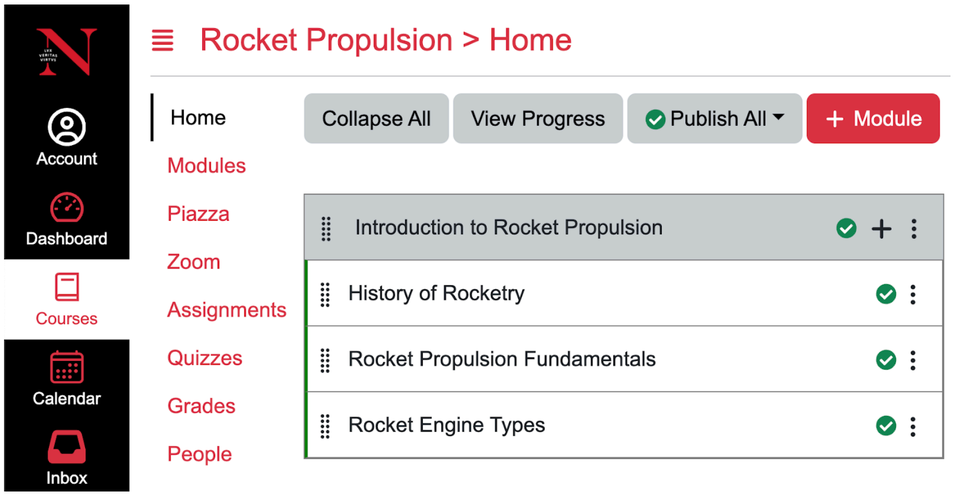
Below is a suggestion on how you could implement the breadcrumbs.

src/Kanbas/Courses/index.tsx

```
import { Navigate, Route, Routes, useParams, useLocation } from "react-router";
export default function Courses() {
  const { cid } = useParams();
  const course = courses.find((course) => course._id === cid);
  const { pathname } = useLocation();
  return (
    <div id="wd-courses">
      <h2 className="text-danger">
        <FaAlignJustify className="me-3 fs-4 mb-1" />
        {course && course.name} &gt; {pathname.split("/")[4]}
      </h2>
    </div>
  );
}
```

3.7 Data Driven Modules Screen

Currently the ***Modules*** and ***Home*** screen render the same course modules regardless which course you click on in the ***Dashboard***. In this section we're going to modify the ***Modules*** screen so that we display the corresponding modules for the selected course. Each course has a different set of modules and each module has its set of lessons. Use the data provided and save it to ***src/Kanbas/Database/modules.json*** file containing at least 3 modules for each course. Include the modules in the ***Database*** as shown below.



```
src/Kanbas/Database/index.ts
```

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

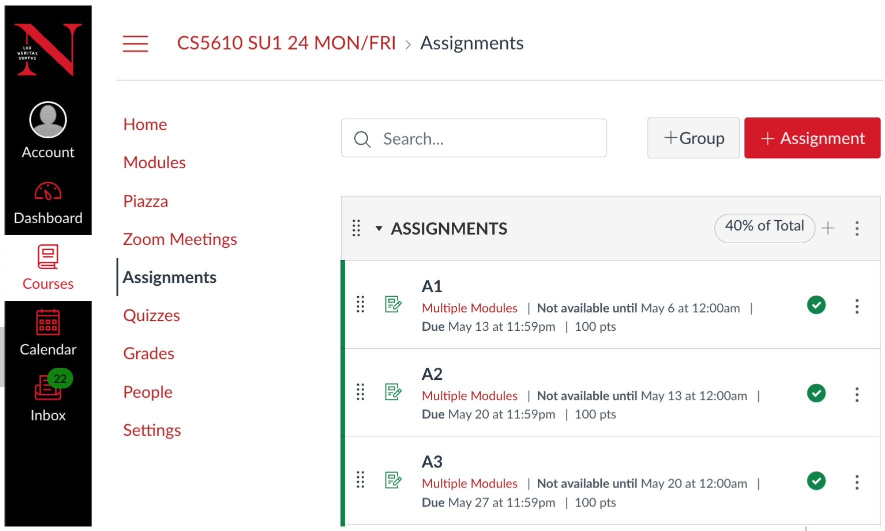
Below is an example of how you can use the ***modules*** in the ***Database*** to render a list of modules. Use ***useParams()*** to retrieve the course ID from the URL and then retrieve the corresponding modules for the course. Confirm that navigating to different courses, the corresponding modules and lessons are rendered as shown above.

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

```
import { useParams } from "react-router";
import * as db from "../../Database";
export default function Modules() {
  const { cid } = useParams();
  const modules = db.modules;
  return (
      ...
      <ul id="wd-modules" className="list-group rounded-0">
        {modules
          .filter((module: any) => module.course === cid)
          .map((module: any) => (
          <li className="wd-module list-group-item p-0 mb-5 fs-5 border-gray">
            <div className="wd-title p-3 ps-2 bg-secondary">
              <BsGripVertical className="me-2 fs-3" />
              {module.name}
              <ModuleControlButtons />
            </div>
            {module.lessons && (
              <ul className="wd-lessons list-group rounded-0">
                {module.lessons.map((lesson: any) => (
                  <li className="wd-lesson list-group-item p-3 ps-1">
                    <BsGripVertical className="me-2 fs-3" />
                    {lesson.name}
                    <LessonControlButtons />
                  </li>
                ))}
              </ul>
            )}
          </li>
        ))}
      </ul>
...);}
```

3.8 Data Driven Assignments Screen (On Your Own)

Based on your implementation of the ***Assignments*** screen in previous assignments, refactor the component so that it renders the assignments for the currently selected course. Use the data provided as an example to create an ***assignments.json*** file which contains various assignments for the courses in the ***courses.json*** file provided. Include the ***assignments*** in the ***Database*** as shown below.



A1
Multiple Modules | Not available until May 6 at 12:00am |
Due May 13 at 11:59pm | 100 pts

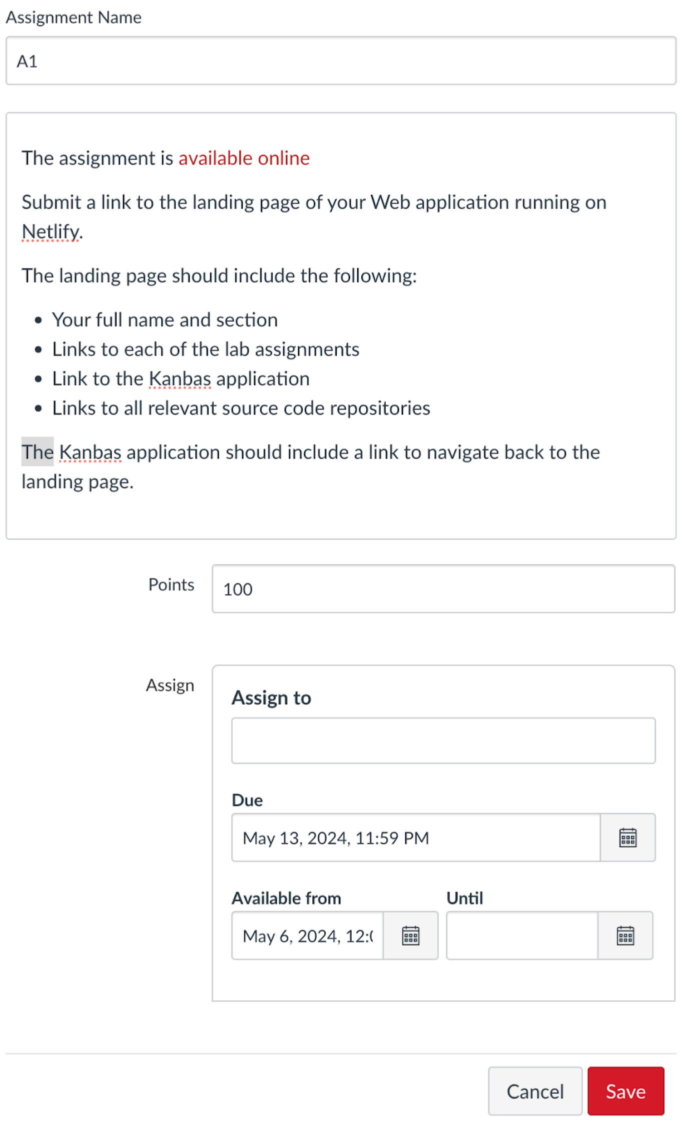
A2
Multiple Modules | Not available until May 13 at 12:00am |
Due May 20 at 11:59pm | 100 pts

A3
Multiple Modules | Not available until May 20 at 12:00am |
Due May 27 at 11:59pm | 100 pts

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

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

Use the ***useParams()*** hook to retrieve the ***course***'s ID and then find all the assignments for that course from the database's ***assignments*** array. Render the assignments as links that encode the ***course***'s ID and the ***assignment***'s ID in the URL's path. The ***assignment***'s ***ID*** will be used by a router to render the corresponding assignment in the ***AssignmentEditor*** screen.



The assignment is available online

Submit a link to the landing page of your Web application running on Netlify.

The landing page should include the following:

- Your full name and section
- Links to each of the lab assignments
- Link to the Kanbas application
- Links to all relevant source code repositories

The Kanbas application should include a link to navigate back to the landing page.

3.8.1 Data Driven Assignment Editor Screen (On Your Own)

Based on the ***Assignment Editor*** screen implemented in earlier assignments, refactor the screen so that it renders the information for the assignment selected in the ***Assignments*** screen. Use ***useParams()*** to parse the ***assignment***'s ***ID***s from the URL and retrieve the assignment from the database's ***assignments*** object. Display the ***title*** of the selected assignment as well as the ***description***, ***points***, ***due date***, and ***available date***. Use ***useParams()*** to parse the ***course***'s ***ID*** from the URL and implement the ***Cancel*** and ***Save*** buttons as ***Link*** so that clicking either one navigates back to the ***Assignments*** screen for the current course. The ***Assignment Editor*** screen should look as shown here on the right. Other fields such as ***Submission Type*** and ***Assignment Group*** are not shown for brevity, but they should still appear as implemented in earlier assignments.

3.9 Data Driven People Screen

Previous assignments implemented and styled the ***People*** screen that displays a list of users as a table. The current implementation is static, always showing the same hard coded list of students. In this section, refactor the ***People*** screen to display students, teaching assistants, and faculty associated with the selected course. Use the data provided as an example list of users and save the file as ***users.json*** file in the ***Database*** folder. Import ***users.json*** into ***Database/index.ts*** so that it can be imported from the ***People/Table.tsx***. 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": "123", "course": "RS101" },
  { "_id": "2", "user": "234", "course": "RS101" },
  { "_id": "3", "user": "345", "course": "RS101" },
  { "_id": "4", "user": "456", "course": "RS101" },
  { "_id": "5", "user": "567", "course": "RS101" },
  { "_id": "6", "user": "234", "course": "RS102" },
  { "_id": "7", "user": "789", "course": "RS102" },
  { "_id": "8", "user": "890", "course": "RS102" },
  { "_id": "9", "user": "123", "course": "RS102" }
]
```

In the ***PeopleTable*** screen, parse the current course ID from the path using ***useParams*** so that the users associated with the course can be filtered based on their enrollment.

```
src/Kanbas/Courses/People/Table.tsx
```

```
import React from "react";
import { useParams } from "react-router-dom";
import * as db from "../../Database";
export default function PeopleTable() {
  const { cid } = useParams();
  const { users, enrollments } = db;
  return (
    <div id="wd-people-table">
      ...
    </div>
  );
}
```

Replace the hardcoded rows displaying the users with an expression that displays only the users related to the currently selected course. The ***users.filter*** expression below filters the array of users in the database. The filter function searches the users in the ***enrollments*** array based on the user's ID and the user's enrollment in the selected course. Navigate to various courses and confirm that only the users enrolled in the courses are shown in the ***People*** screen.

```
src/Kanbas/Courses/People/Table.tsx
```

```
<tbody>
  {users
    .filter((usr) =>
      enrollments.some((enrollment) => enrollment.user === usr._id && enrollment.course === cid)
    )
    .map((user: any) => (
      <tr key={user._id}>
        <td className="wd-full-name text-nowrap">
          <FaUserCircle className="me-2 fs-1 text-secondary" />
          <span className="wd-first-name">{user.firstName}</span>
          <span className="wd-last-name">{user.lastName}</span>
        </td>
        <td className="wd-login-id">{user.loginId}</td>
        <td className="wd-section">{user.section}</td>
        <td className="wd-role">{user.role}</td>
        <td className="wd-last-activity">{user.lastActivity}</td>
        <td className="wd-total-activity">{user.totalActivity}</td>
      </tr>
    ))}
</tbody>
```

4 Deliverables

1. In the same React.js application created in earlier assignments, ***kanbas-react-web-app***, complete all the exercises described in this document
2. In a branch called ***a3***, add, commit and push the source code of the React.js application ***kanbas-react-web-app*** to the same remote source repository in ***GitHub.com*** created in an earlier assignment. Here's an example of how to add, commit and push your code

```
$ git   checkout  -b  a3
$ git   add   .
$ git   commit   -am   "a3 JavaScript"
$ git   push
```

3. Deploy the ***a3*** branch to the same ***Netlify*** project created in an earlier assignment. Configure Netlify to deploy all branches to separate URLs. From your Netlify's dashboard go to ***Site settings > Build & deploy > Branches > Branch deploys*** and select ***All***. Now each time you commit to a branch, the application will be available at a URL that contains the name of the branch

4. Make sure ***Labs/index.tsx*** contains a ***TOC.tsx*** that references each of the labs and Kanbas. Add a link to your repository in GitHub. The link should have an ***ID*** attribute with a value of ***wd-github***.
5. In ***Labs/index.tsx***, add your full name: first name first and last name second. Use the same name as in Canvas.
6. Deploy the branch for this assignment to your ***kanbas-react-web-app*** React.js application to ***Netlify*** as described.
7. As a deliverable in ***Canvas***, submit the URL to the ***a3*** branch deployment of your React.js application running on Netlify.