Creating Dynamic Web Pages with JavaScript

1 Introduction

HTML and CSS on their own can create compelling static Web pages. We'll need a programming language to build applications that can interact with users, respond to their input, and update the user interface accordingly. We'll be using JavaScript as the programming language to program the browser. We'll write JavaScript code that can react to user events, can generate dynamic HTML, and modify the DOM programmatically. In this assignment we'll take the static Web pages from previous assignments and refactor them to use JavaScript.

2 Labs

This section contains several ***JavaScript*** and ***jQuery*** exercises that will give us a chance to practice programming the browser, interacting with the user, and generating dynamic HTML. Use the same project you worked on last assignment. After you work through the exercises you will apply the skills to create ***Tuiter*** on your own.

Using ***IntelliJ***, open the project you created in the previous assignment. From within IntelliJ, use ***File***, ***Open Project***, and navigate to the project directory, (***tuiter-react-web-app***), and click ***Open*** or ***OK***. From within IntelliJ, on the ***Project*** tab, open the ***tuiter-react-web-app*** directory, and then the ***public*** directory. Do all your work under the ***public*** directory of your project. Under the ***public/labs*** directory, create a new directory called ***a5*** and create ***index.html*** under ***public/labs/a5***. For this assignment, do all your work in ***public/labs/a5/index.html***. To make it easy on the TAs to find your assignment, add a link to this new ***index.html*** file in ***public/index.html***.

2.1 Embedding JavaScript Programs in HTML Documents

Scripts can be added to HTML documents by embedding them in a ***script*** tag as shown below. Copy the code below into your ***index.html***, refresh and confirm an ***alert*** display. Browsers immediately execute all scripts. After confirming the alert worked, comment out the script.

```
<h2>Embedded script tag</h2>
<script>
   alert('Hello World!');
</script>
```

2.2 Loading External JavaScript Files

Instead of embedding scripts right in the HTML document, a better alternative is to write the scripts in separate files and load them using the scripts tag. To practice this create a new file called ***index.js*** and copy the code below.

```
alert('Hello World!');
```

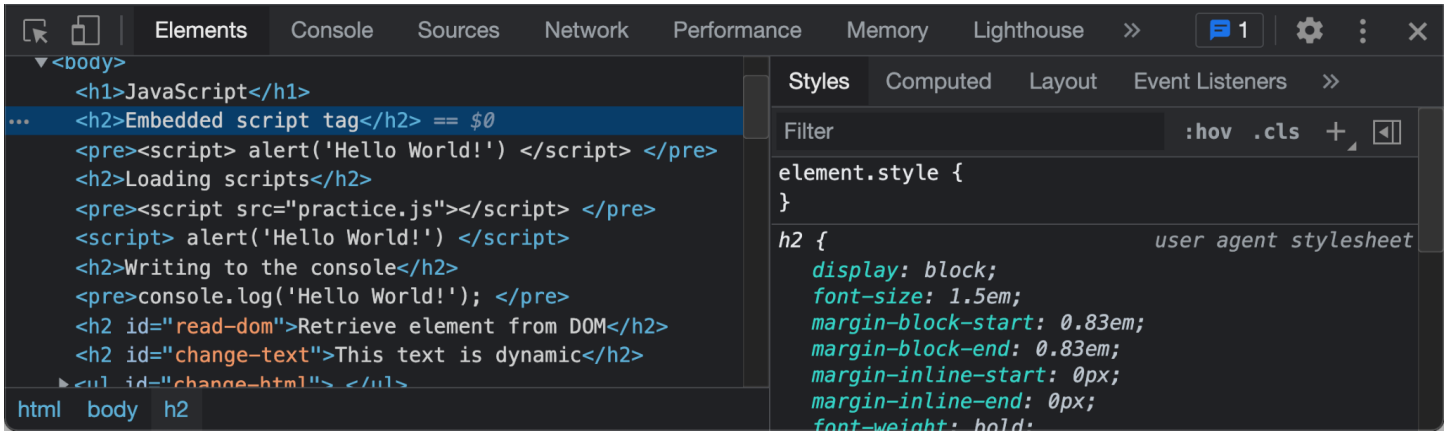
Then load the ***index.js*** file from the ***index.html*** file as shown below. Paste it right before the closing ***</body>*** tag as shown. Make sure to keep this ***<script>*** tag at the end of the file. Also remove the original ***<script>*** tag embedding the ***alert*** command.

```
<body>
<h2>Embedded script tag</h2>
<script>
    alert('Hello World!');
</script>
<!-- paste the rest of the exercises here before the script tag shown here below -->
<script src="index.js"></script>
</body>
```

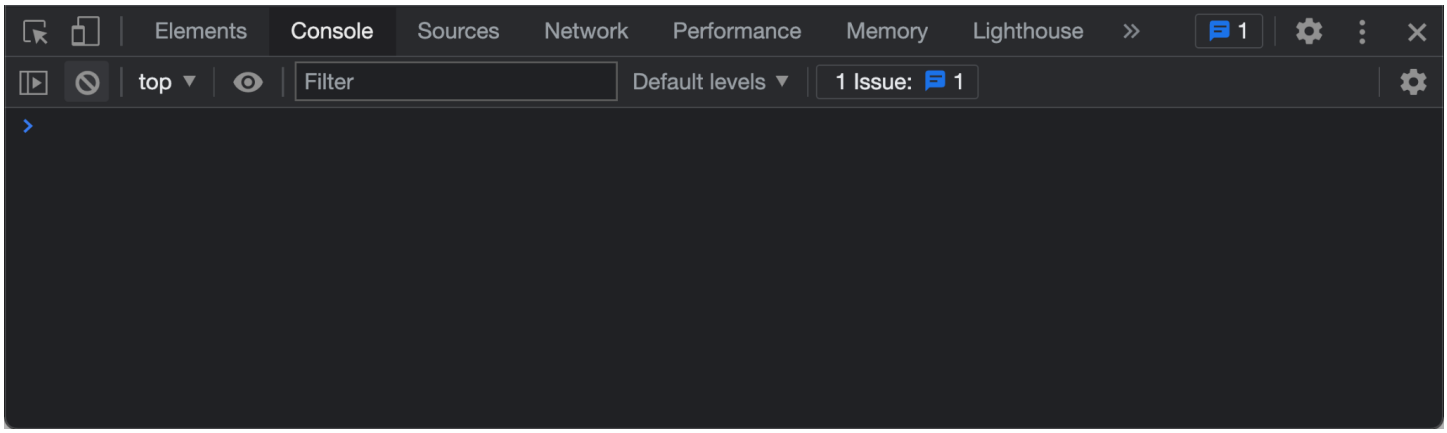
Reload and confirm the alert pops up again. Once done, comment out the alert command so it won't keep popping up as you work on the rest of the exercises.

2.3 Writing to the Console

A useful feature of modern browsers is that they provide a development environment where you can analyze the performance of your scripts. One way to analyze if your script is behaving correctly is to write output to the ***console*** from within your script. To practice writing to the ***console***, bring up the console on your 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.



Copy and paste the following to the end of ***index.js*** and then reload the screen. Confirm that "***Hello World!***" is displayed in the console.

index.js	***Console***
~~*alert*('Hello World!');~~ *console*.log('Hello World!');	Hello World!

2.4 Variables and Constants

Variables enable storing state information about applications. To practice declaring variables and constants, copy and paste the code below to the end of ***index.js***. 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 TAs will be looking at the console output to verify your code is working as expected.

index.js	***Console***

console.log('Variables and Constants');
global1 = 10;
var *functionScoped* = 2;
let *blockScoped* = 5;
const *constant1* = global1
+ *functionScoped*
- *blockScoped*;
	Variables and Constants index.js:4
	10 index.js:11
	2 index.js:12
	5 index.js:13
	7 index.js:14

2.4.1 Variable Types

JavaScript declares several datatypes such as Number, String, Date, and so on. To practice with variable types, copy and paste the code below to the end of ***index.js***. 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 TAs will be looking at the console output to verify your code is working as expected.

index.js	***Console***

```
console.log('Variable types');
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;
```

```
Variable types          index.js:17
123                     index.js:26
234.345                 index.js:27
Hello World!            index.js:28
true                    index.js:29
number                  index.js:30
string                  index.js:31
boolean                 index.js:32
```

2.4.2 Boolean Variables

To practice with Boolean data types, copy and paste the code below to the end of ***index.js***. 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.

index.js

```
console.log('Boolean Variables');
let true1 = true;
let false1 = false;
let false2 = true1 && false1;
let true2 = true1 || false1;
let true3 = !false2;
let true4 = numberVariable === 123;
let true5 = floatingPointNumber !== 321.432;
let false3 = numberVariable < 100;
let sortaTrue = '1' == 1
let notTrue   = '1' === 1
```

Console

```
Boolean Variables       index.js:34
true                    index.js:44
false                   index.js:45
false                   index.js:46
true                    index.js:47
true                    index.js:48
true                    index.js:49
true                    index.js:50
false                   index.js:51
```

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.

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, copy and paste the code below to the end of ***index.js***. Confirm the console displays as shown on the right below. The TAs will be looking at the console output to verify your code is working as expected.

index.js ***Console***

```
console.log('If else');
if(true1) {
   console.log(true);
}

if(!false1) {
   console.log('!false1');
} else {
   console.log('false1');
}
```

```
If else                    index.js:53
true                       index.js:55
!false1                    index.js:59
```

2.5.2 Ternary Conditional Operator

Ternary conditional operators are concise alternative to the if statement. It takes three arguments: a conditional expression that evaluates to true or false followed by a question mark (?). Then an expression that evaluates if the conditional is true followed by a colon (:), followed by an expression that evaluates iff the conditional is false. To practice, copy the code below and confirm the console displays as shown. Ignore extra spaces shown in the console below. They were added to align code on the left with output on the right.

index.js	*Console*

`console.log('Ternary conditional operator');`
`const loggedIn = true;`
`const greeting = loggedIn ? 'Welcome!' : 'Please login';`
`console.log(greeting)`
	`Ternary conditional operator`
	`Welcome!`

2.6 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.

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, copy the code below, refresh the page, and confirm the console output. Again, ignore the extra spaces on the console on the right. They were added to align code on the left with output on the right. Print the name of each section so TAs know what output belongs to what exercise, e.g., "Legacy ES5 function"

index.js	*Console*
`console.log('Legacy ES5 function')` `function add (a, b) {`	`Legacy ES5 function`

```
   return a + b;
}
const twoPlusFour = add(2, 4);
console.log(twoPlusFour);
```

```
6
```

2.6.2 New 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, copy the code below, refresh the page, and confirm the console output.

index.js

```
console.log('New ES6 functions')
const subtract = (a, b) => {
   return a - b;
}
const threeMinusOne = subtract(3, 1);
console.log(threeMinusOne);
```

Console

```
New ES6 functions


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 copy the code below, refresh, and confirm the console output.

index.js

```
const multiply = (a, b) => a * b;
const fourTimesFive = multiply(4, 5);
console.log(fourTimesFive);
```

Console

```
Implied return

20
```

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 copy the code below, refresh and confirm the console output.

index.js

```
const square = a => a * a;
const plusOne = a => a + 1;
const twoSquared = square(2);
const threePlusOne = plusOne(3);
console.log(twoSquared);
console.log(threePlusOne);
```

Console

```
Parenthesis and parameters


4
4
```

2.7 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.js***. 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.

index.js	***Console***

`let numberArray1 = [1, 2, 3, 4, 5];`
`let stringArray1 = ['string1', 'string2'];`
`let variableArray1 = [`
`  functionScoped,`
`  blockScoped,`
`  constant1,`
`  numberArray1,`
`  stringArray1`
`];`
	Arrays
	(5) [1, 2, 3, 4, 5]
	(2) ["string1", "string2"]
	(5) [2, 5, 7, Array(5), Array(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, copy and paste the code below to the end of ***index.js***. 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.

index.js	***Console***
`const length1 = numberArray1.length;` `const index1 = numberArray1.indexOf(3);`	***Array index and length*** 5 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, copy and paste the code below to the end of ***index.js***. 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.

index.js	***Console***

`// adding new items`
`numberArray1.push(6);`
`stringArray1.push('string3');`
	Add and remove data to arrays

`// remove 1 item starting on 3rd spot`
`numberArray1.splice(2, 1);`
`stringArray1.splice(1, 1);`
`console.log(numberArray1);`
`console.log(stringArray1);`
	[1, 2, 4, 5, 6]
	["string1", "string3"]

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, copy and paste the code below to the end of ***index.js***. Confirm the console displays as shown on the right below.

index.js	***Console***

`for (let i=0; i<stringArray1.length; i++) {`
`  const string1 = stringArray1[i];`
`  console.log(string1);`
`}`
	For loops
	string1
	string3

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***, copy and paste the code below to the end of ***index.js***. Use ***console.log()*** to print the title of this section and then ***numberArray1***, ***squares***, and ***cubes***. Confirm the console displays as shown on the right below.

index.js	***Console***
`const squares = numberArray1.map(square);` `const cubes = numberArray1.map(a => a * a * a);`	***Map function*** (5) [1, 4, 16, 25, 36] (5) [1, 8, 64, 125, 216]

2.7.5 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 another 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, copy the code below to the end of ***index.js***. 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.

index.js	*Console*
`const four = numberArray1.find(a => a === 4);` `const string3 = stringArray1.find(a => a === 'string3');`	***Find function*** 4 string3

2.7.6 The Find Index Function

Alternatively we can use ***findIndex*** function to determine the index where an element is located inside an array. To practice, copy the code below to the end of ***index.js***. 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.

index.js	*Console*

`const fourIndex = numberArray1`
`  .findIndex(a => a === 4);`
`const string3Index = stringArray1`
`  .findIndex(a => a === 'string3');`
	Find index
	2
	1

2.7.7 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, copy the code below to the end of ***index.js***. Use ***console.log()*** to print the title of this section and then ***numberArray1***, ***numbersGreaterThan2***, ***evenNumbers***, and ***oddNumbers***. Confirm the console displays as shown on the right below.

index.js	*Console*

`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
	
	(3) [4, 5, 6]
	(3) [2, 4, 6]
	(2) [1, 5]

2.8 Template Strings

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, copy the code below to the end of ***index.js***. Use ***console.log()*** to print the title of this section and the rest of the variables. Confirm the console displays as shown.

index.js	*Console*

`const five = 2 + 3;`
`const result1 = "2 + 3 = " + five;`
`console.log(result1);`
 	
`const result2 = `2 + 3 = ${2 + 3}`;`
`console.log(result2);`
 	
`const username = "alice";`
`const greeting1 = `Welcome home ${username}`;`
`console.log(greeting1);`
 	
`loggedIn = false;`
`const greeting2 = `Logged in: ${loggedIn ? "Yes" : "No"}`;`
`console.log(greeting2)`
	Template strings
	
	2 + 3 = 5
	
	2 + 3 = 5
	
	Welcome home alice
	
	Logged in: No

2.8.1 Writing to the DOM

All the exercises we've gone through so far have written their output to the console. Although useful, it's not nearly as exciting as writing to the ***DOM*** (***Document Object Model***). JavaScript can manipulate the DOM, the browser's in memory representation of an HTML document. New content can be created writing HTML strings to the DOM. The DOM is represented in JavaScript by the global variable document. To practice, copy the code below to the end of ***index.js***, refresh the browser and confirm it displays as shown.

index.html	*Browser*
`<h2>Writing to the DOM</h2>` `<script>` `  document.write('Welcome to JavaScript!')` `</script>` `<h2>Calculating an HTML string</h2>` `<script>` `  const renderTodos = (todos) => ` `    <ul>` `      ${todos.map(todo =>` `       ` ${todo} ` `).join('')}` ` `;` `  const todos = ['Pickup kids', 'Feed dogs']` `  let html = renderTodos(todos);` `  document.write(html);` `</script>`	**Writing to the DOM** Welcome to JavaScript! **Calculating an HTML string** • Pickup kids • Feed dogs

2.9 Loading jQuery library

Although it is entirely possible to use pure JavaScript to build fully functional Web applications, it is significantly easier if we use libraries such as ***jQuery*** which provide tons of great prebuilt functions that simplify interacting with the DOM. The ***jQuery*** library can be installed by loading it from a ***CDN*** as shown below.

index.html

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
```

Linking scripts from a CDN is preferable in production environments. Alternatively you can download the library into your local vendors directory and then link it as shown below. Make sure to add the library before the existing ***index.js*** since it will require variables and functions declared by the jQuery library.

index.html

```
<script src="../../vendors/jquery/jquery-3.6.0.min.js"></script>
<script src="index.js"></script>
```

2.10 Executing jQuery scripts

The ***jQuery*** script declares global function called ***jQuery*** and its alias ***$***. The ***jQuery*** function can manipulate the DOM in many different ways, but a common practice is to pass it an initialization function. To practice intializing ***jQuery***, copy and paste the code below to the end of ***index.js***. Confirm the console displays as shown on the right below.

index.js

```
const init = () => {
   console.log('Hello world from jQuery');
   /* do the rest of the lab work here */
}
$(init);
```

Console

Hello world from jQuery

What's happening here is that ***jQuery*** waits for the document to load, the DOM to be generated, and then it invokes the initialization function. The function doesn't need to be called ***init***, but it is a common name.

2.11 Binding to the DOM

One of jQuery's strength is to bind to the DOM by referring to different parts using CSS selectors. The examples below reference DOM elements in the left using CSS strings. For instance, the ***H2#bind-by-id*** heading is referenced with ***$('#bind-by-id')*** and storing it in a variable for further processing. To practice with binding to the DOM, copy and paste the code below to the end of the ***init()*** function. 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.

index.html

```
<h2 id="bind-by-id">
   Binding by ID</h2>
<h3 class="bind-by-class">
   Binding 1 by Class</h3>
```

index.js

```
const bindById = $('#bind-by-id');
const bindByClass =
$('.bind-by-class');
console.log('Binding to DOM');
```

Console

Binding to the DOM
(1) [h2#bind-by-id]
(2) [h3.bind-by-class,

`<h3 class="bind-by-class">` `  Binding 2 by Class</h3>`	`console.log(bindById);` `console.log(bindByClass);`	h3.bind-by-class]

2.12 Changing style programmatically

Once jQuery binds to a DOM element, we can use all sort of methods to manipulate the DOM element. In this example we modify a DOM's style using the ***css*** method. To practice changing style programmatically, copy and paste the code below to the end of the ***init()*** function. Confirm the browser displays as shown on the right below.

index.html	***index.js***	***Browser***

`<h2 id="change-style">`
`  Change style</h2>`
`<h3 class="change-style">`
`  Change style 1</h3>`
`<h3 class="change-style">`
`  Change style 2</h3>`
	`const changeStyle = $('#change-style');`
	`const changeStyle2 = $('.change-style');`
	`changeStyle.css('color', 'red');`
	`changeStyle2.css('color', 'blue');`
		Change style
		Change style 1
		Change style 2

2.13 Getting and setting attributes

We can also modify an element's attributes by using the ***attr*** method. The example below reads the ***ID*** attribute of the heading element. To practice getting an attribute, copy and paste the code below to the end of the ***init()*** function. Confirm the console displays as shown.

index.html	***index.js***	***console***

`<h2 id="get-id-attr">`
`  Get id attribute`
`</h2>`
	`const getIdAttr =`
	`$("#get-id-attr");`
	`const id = getIdAttr.attr('id');`
	`console.log(id);`
		Get and set attributes
		get-id-attr

The ***attr*** function can not only be used to read the value of an attribute like above, but it can also be used to modify its value. The example below sets the value of the ***class*** attribute. To practice setting an attribute, copy and paste the code below to the end of the ***init()*** function. You'll need to create a brand new ***index.css*** under ***public/labs/a4*** and then link it in ***index.html*** as described in previous labs. Confirm the console displays as shown.

index.css	***index.hthml***

`.class-0 {`
`  background-color: red;`
`  color: white;`
`}`
	`<h2 id="set-class-attr">Set class attribute</h2>`

index.js	***Browser***

```
const setClassAttr = $("#set-class-attr");
setClassAttr.attr('class', 'class-0');
```

Set class attribute

2.14 Adding and removing classes

Modifying an element's ***class*** attribute is such a common task that jQuery provides dedicated methods ***addClass*** and ***removeClass***. The example below adds class ***class-1*** to the heading. To practice adding classes programmatically, copy and paste the code below to the end of the ***init()*** function. Confirm the browser displays as shown on the right below.

index.css

```
.class-1 {
   background-color: blue;
   color: white;
}
```

index.html

```
<h2 id="add-class-1">Add class</h2>
```

index.js

```
const addClass1Example = $("#add-class-1");
addClass1Example.addClass('class-1');
```

Browser

Add class

The example below removeds the class ***class-2***. To practice removing classes programmatically, copy and paste the code below to the end of the ***init()*** function. Confirm the browser displays as shown on the right below.

index.css

```
.class-2 {
   background-color: green;
   color: yellow;
}
```

index.hthml

```
<h2 id="remove-class-1" class="class-2">
   Remove class
</h2>
```

index.js

```
const removeClass1Example = $("#remove-class-1");
removeClass1Example.removeClass('class-2');
```

Browser

Remove class

2.15 Hiding and showing content

The ***jQuery*** library provides methods ***hide*** and ***show***. To practice hiding content programmatically, copy and paste the code below to the end of the ***init()*** function. Confirm the browser does not show the Hide me header.

index.hthml

```
<h2 id="hide-me">Hide me</h2>
```

index.js

```
const hideMe = $("#hide-me");
hideMe.hide();
```

To practice showing content programmatically, copy and paste the code below to the end of the ***init()*** function. Confirm the browser displays as shown on the right below.

index.css	*index.hthml*
`.im-hidden { display: none; }`	`<h2 id="show-me" class="im-hidden">Show me</h2>`
index.js	***Browser***
`const showMe = $("#show-me"); showMe.show();`	**Show me**

2.16 Creating new elements

Beyond binding to already existing DOM elements, ***jQuery*** can also create brand new elements and add them to the DOM. To create an element, provide an HTML string to the ***jQuery*** or ***$*** function. To practice changing style programmatically, copy and paste the code below to the end of the ***init()*** function. Confirm the browser displays as shown on the right below. Creating the elements does not automatically add them to the DOM.

index.js
`const newLineItem = $("<li>Line item 1</li>"); const anotherLineItem = $("<li>Line item 2</li>");`

2.17 Appending new elements

To actually add new content to the DOM use functions such as ***append***. To practice appending new elements programmatically, copy and paste the code below to the end of the ***init()*** function. Confirm the browser displays as shown on the right below.

index.html	*index.js*	*Browser*
`<h2>Appending new elements</h2> <ul id="append-new-elements"> </ul>`	`const ul = $("#append-new-elements"); ul.append(newLineItem); ul.append(anotherLineItem);`	• Line item 1 • Line item 2

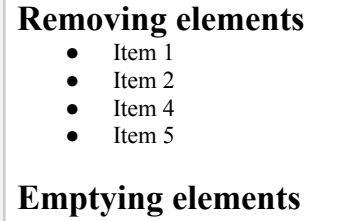
2.18 Removing and emptying content

We can also remove elements from DOM. To practice removing and emptying content, copy and paste the code below to the end of the ***init()*** function. Confirm the browser displays as shown on the right below.

index.html	*index.js*	*Browser*

```html
<h2>Removing elements</h2>
<ul>
   <li>Item 1</li>
   <li>Item 2</li>
   <li id="remove-this">Item 3</li>
   <li>Item 4</li>
   <li>Item 5</li>
</ul>

<h2>Emptying elements</h2>
<ul id="empty-this">
   <li>Item A</li>
   <li>Item B</li>
   <li>Item C</li>
   <li>Item D</li>
   <li>Item E</li>
</ul>
```

```js
const removeLi =$("#remove-this");
const emptyUl = $("#empty-this");
removeLi.remove();
emptyUl.empty();
```



2.19 Changing content

We can also modify content already in the DOM. To practice changing content, copy and paste the code below to the end of the ***init()*** function. Confirm the browser displays as shown on the right below.

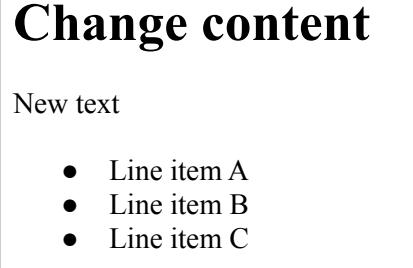
index.html

```html
<h2>Change content</h2>
<p id="change-this-text">
   Change me
</p>
<ul id="change-this-html">
</ul>
```

index.js

```js
const changeThisText =
   $("#change-this-text");
const changeThisHtml =
   $("#change-this-html");
changeThisText.html('New text');
changeThisHtml.html(`
   <li>Line item A</li>
   <li>Line item B</li>
   <li>Line item C</li>
`);
```

Browser



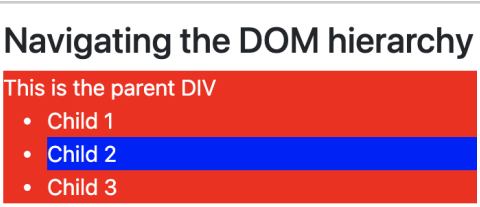
2.20 Navigating up and down the DOM tree

We often need to refer to DOM elements that are nested within another DOM or is the parent of some element. The ***parents()*** function returns an array of elements that are parents of an element all the way up to the root of the HTML document. The ***find()*** function returns an array of elements that are children of the current element. Both accept CSS selectors to filter parents or children that match the selector. To practice navigating the DOM hierarchy, copy and paste the code below to the end of the ***init()*** function. Confirm the browser displays as shown on the right below.

index.html | ***index.js*** | ***Browser***

```
<h2>Navigating the
   DOM hierarchy</h2>
<div id="parent">
   This is the parent DIV
   <ul>
       <li>Child 1</li>
       <li
id="child-2">Child 2</li>
       <li>Child 3</li>
   </ul>
</div>
```

```
const child2 = $("#child-2");
const parent1 =
   child2.parents("#parent");
parent1
   .css('background-color',
'red')
   .css('color', 'white');

const parent = $("#parent");
const child =
parent.find("#child-2");
child.css('background-color',
'blue')
```



2.21 Handling click events

The ***jQuery*** library can handle all sorts of events generated by the user. The most common event is clicking the mouse. To practice handling click event, copy and paste the code below to the end of the ***init()*** function. Confirm the browser displays as shown on the right below and the console displays ***Handle click***.

index.html

```
<h2>Event handling</h2>
<h2 class="clickable">
   Click event</h2>
<p class="clickable">
   Anything can be
clickable</p>
<button class="clickable">
   Not just me</button>
```

index.js

```
const handleClick =
   () => console.log('Handle
click');
const clickable =
$('.clickable');
clickable.click(handleClick);
```

Browser

2.22 Event target

We often need to determine the UI element that a user clicked on. This is available in the ***target*** attribute of the ***event*** generated by the user. To practice working with the event's target, copy and paste the code below to the end of the ***init()*** function. Confirm the browser displays as shown on the right below. Confirm clicking heading changes color.

index.html

```
<h2 id="event-target">
   Event target
</h2>
```

index.js

```
const handleEventTarget = (event) => {
   const target = event.target;
   console.log(target);
   $(target)
       .css('background-color', 'blue')
       .css('color', 'white');
}
const eventTarget = $("#event-target");
eventTarget.click(handleEventTarget);
```

Browser

2.23 Hiding and showing content

We can combine event handlers with DOM manipulation to achieve useful behaviors. In this example we handle click events to either hide or show DOM elements. To practice hiding and showing content, copy and paste the code below to the end of the ***init()*** function. Confirm the browser displays as shown on the right below.

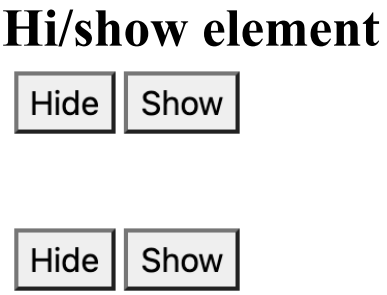
index.html

```
<h3 id="hide-show">
   Hi/show element
</h3>

<button id="hide">
   Hide
</button>

<button id="show">
   Show
</button>
```

index.js

```
let hideBtn, showBtn, hideShowHeader;
hideBtn = $('#hide');
showBtn = $('#show');
hideShowHeader = $('#hide-show');
const hideHandler = () => {
   hideShowHeader.hide();
}
const showHandler = () => {
   hideShowHeader.show();
}
hideBtn.click(hideHandler);
showBtn.click(showHandler);
```

Browser



2.24 Creating a TODO List

To practice using ***JavaScript*** and ***jQuery***, let's create a simple todo list. To start, create folder called ***public/labs/a5/todos***, and create an ***index.html*** file with the HTML shown below. Add a link to this new file in ***public/index.html*** so TAs can find it for grading

```
<!DOCTYPE html>
<html lang="en">
<head>
   <title>TODO List</title>
</head>
<body>
</body>
</html>
```

In the ***head*** tag, add ***Bootstrap***, ***Fontawesome***, and ***jQuery*** as shown below highlighted in yellow. Your folder locations might differ.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <link href="../../../vendors/bootstrap/css/bootstrap.min.css" rel="stylesheet"/>
   <link href="../../../vendors/fontawesome/css/all.min.css" rel="stylesheet"/>
   <script src="../../../vendors/jquery/jquery-3.6.0.min.js"></script>
   <title>TODO List</title>
</head>
<body>
</body>
</html>
```

In the same folder, create a new ***index.js*** file and load it in the ***body*** of the ***index.html*** file as shown below. Refresh the screen and confirm that a popup appears saying ***Hello World!***.

index.html

```
<body>
<!-- do your work here -->
<script src="index.js" type="module"></script>
</body>
```

index.js

```
alert('Hello World!');
```

Right before loading the ***index.js*** file, add a ***div*** with an ***id*** attribute set to ***wd-todos***. We'll dynamically render all the content in this ***div***. In the ***index.js*** file, replace the content with the code below which ***appends*** HTML into the new ***div***. Refresh the page and confirm it displays the new content. Note that we are using back ticks (`) inside the append function, not single quotes ('). The relevant point here is that we are programmatically adding content into the DOM, instead of using static HTML.

index.html

```
<body>
<div id="wd-todo"></div>
<script src="index.js" type="module"></script>
</body>
```

index.js

```
~~alert('Hello World!');~~
$('#wd-todo').append(`
   <div class="container">
       <h1>Todo example</h1>
   </div>
`);
```

Create a new file called ***TodoList.js*** and import it into ***index.js***. Use the ***TodoList*** function to render the list of todos as shows below. Refresh the page and confirm that the list renders as expected. What's relevant here is that we are wrapping some static HTML code with a function -- ***TodoList***. This enables us breaking up pages into modules that can then be combined programmatically. Also, functions can use logic to generate complex user interfaces. We often refer to these as ***components***.

index.js

```
import TodoList from "./TodoList.js";

$('#wd-todo').append(`
   <div class="container">
       <h1>Todo example</h1>
       ${TodoList()}
   </div>
`);
```

TodoList.js

```
const TodoList = () => {
   return(`
       <ul>
           <li>Buy milk</li>
           <li>Pickup the kids</li>
           <li>Walk the dog</li>
       </ul>
   `);
}
export default TodoList;
```

Let's create a component out of the line items in the TodoList. We can then parameterize it with the todo's title. Create a new file called ***TodoItem.js*** and import it into ***TodoList.js***. Use the ***TodoItem*** function to render three todo items as shown below. Refresh the page and confirm you have the same todo list. Note the TodoItem function takes as a parameter ***todo*** which is a string we use to render a line item.

TodoList.js

TodoItem.js

```
import TodoItem from "./TodoItem.js";
const TodoList = () => {
   return(`
       <ul>
           ${TodoItem('Buy milk')}
           ${TodoItem('Pickup the kids')}
           ${TodoItem('Walk the dog')}
       </ul>
   `);
}
export default TodoList;
```

```
const TodoItem = (todo) => {
   return(`
       <li>${todo}</li>
   `);
}
export default TodoItem;
```

Instead of invoking ***TodoItem*** several times, we could have all the todos stored in a data structure, like an array, and then iterate over it to render our content. Create a file called ***todos.js*** that contains an array of todos. Import the file from ***TodoList.js*** and use it to render the todos as shown below. Refresh the page and confirm the same todos are rendered. Note that ***TodoList*** is now iterating over the todos array imported from the ***todos.js*** file. We use the function ***map*** to iterate over the ***todos*** array. The ***map*** function iterates over the todos array and for each element in the array, it invokes the functional parameter. Each element is bound to the ***todo*** function parameter, which is used to invoke the ***TodoItem*** function. The ***map*** function collates all the resulting HTML strings from ***TodoItem*** into an array which we then ***join*** them all into a single string.

TodoList.js

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

todos.js

```
export default [
   'Buy milk',
   'Pickup the kids',
   'Walk the dog'
];
```

Let's make things a bit more interesting. Convert the elements of the array in ***todos.js*** into objects with properties ***title*** and ***status***. Update ***TodoItem*** to render the properties as shown below. Refresh the page and confirm the todos now render both the ***title*** and the new ***status*** property in parenthesis.

TodoItem.js

```
const TodoItem = (todo) => {
   return(`
       <li>
           ${todo.title}
           (${todo.status})
       </li>
   `);
}
export default TodoItem;
```

todos.js

```
export default [
   {   title: 'Buy milk',
       status: 'CANCELED',
   },
   {   title: 'Pickup the kids',
       status: 'IN PROGRESS',
   },
   {   title: 'Walk the dog',
       status: 'DEFERRED',
```

	},];

Add boolean attribute ***done*** to the objects and render it as a checkbox in the ***TodoItem.js*** as shown below. Refresh the page and confirm that the checkboxes render as expected.

TodoItem.js

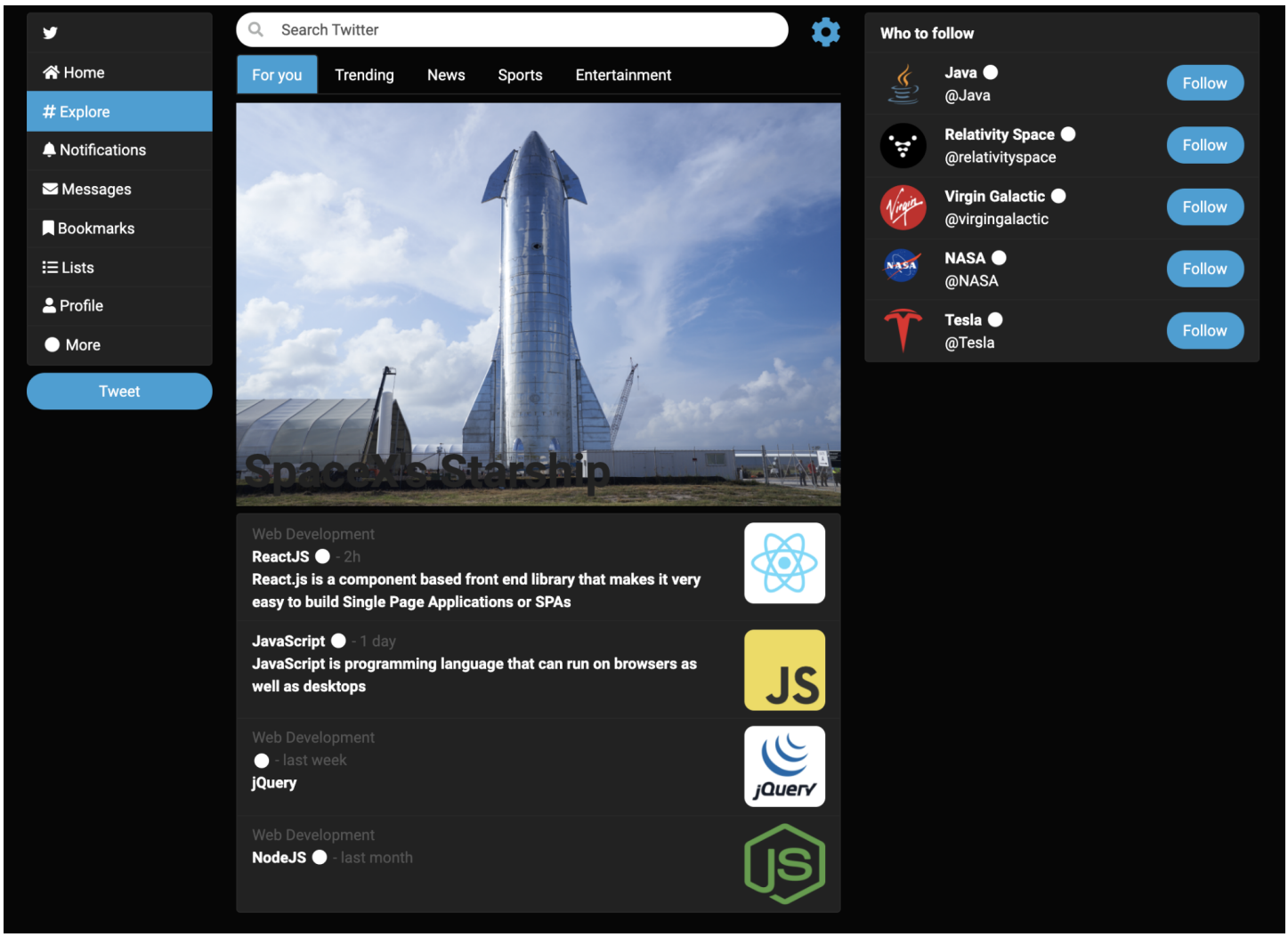
```
const TodoItem = (todo) => {
  return(`
    <li>
      <input type="checkbox"
        ${todo.done ? 'checked' : ''}/>
      ${todo.title}
      (${todo.status})
    </li>`
  );
}
export default TodoItem;
```

todos.js

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

3 Tuiter

For the Tuiter part of this assignment we're going to revisit the ***explore*** screen we worked on in previous assignments. We're going to refactor the static Web page into a set of JavaScript components. This will give us the opportunity to practice JavaScript while introducing reusable components. The resulting screen must look as shown below. You might need to remove some custom styling of your own.



New explore screen

3.1 Bootswatch

Bootswatch is a collection of stylesheets based on Bootstrap. We'll install one of their opensource stylesheets to customize the look and feel a bit. Take a look at the Cyborg theme going through all the examples. Download the theme by clicking Cyborg and then bootstrap.min.css. Move the new stylesheet into a new folder ***public/vendors/bootswatch/***. In the new implementation of the explore screen, link the bootstrap CSS library in the head tag, and then link the bootswatch CSS library right after it so that it overrides the base bootstrap look and feel.

3.2 Explore screen

In a previous assignment we worked on Tuiter's explore screen and implemented it in a file called ***public/tuiter/explore/index.html***. The screen consists of three columns as shown in the picture above. The left column is a navigation sidebar. The right column is a sidebar suggesting other users who you might want to follow. The center column is the main content featuring posts you might be interested in. Previous assignments implemented this screen as a single HTML document. This assignment will refactor it into several JavaScript UI components that will be combined into a single screen. Let's start with reworking the content of the HTML document ***explore/index.html***. Since most of the content is going to live in separate JavaScript components, the resulting ***explore/index.html*** is fairly empty as shown below. We'll implement each screen and component in their dedicated folders. Replace the content of ***explore/index.html*** with the code below.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <link href="../../vendors/bootstrap/css/bootstrap.min.css" rel="stylesheet"/>
   <link href="../../vendors/bootswatch/bootstrap.min.css" rel="stylesheet"/>
   <link href="index.css" rel="stylesheet"/>
   <link href="../../vendors/fontawesome/css/all.css" rel="stylesheet"/>
   <script src="https://code.jquery.com/jquery-3.6.1.min.js"
           integrity="sha256-o88AwQnZB+VDvE9tvIXrMQaPlFFSUTR+nldQm1LuPXQ="
           crossorigin="anonymous"></script>
   <meta charset="UTF-8">
   <title>Explore screen</title>
</head>
<body>

<div class="container" id="wd-explore"></div>
<script src="index.js" type="module"></script>

</body>
</html>
```

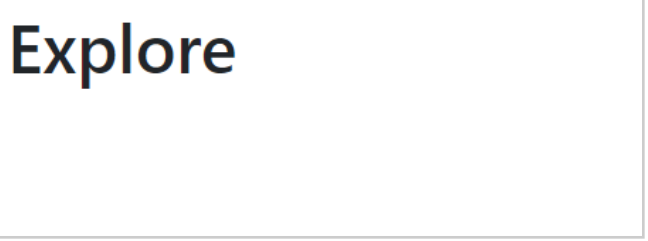
Note that the only real content here is the empty ***div#wd-explore*** element and the ***index.js*** script below it. The script will dynamically generate all the content that will be displayed in ***div#wd-explore***.

3.3 Implement the Explore screen component

Create the JavaScript implementation of the explore component in ***explore/index.js*** as show below. Load ***explore/index.html*** from a browser and confirm the Explore heading renders as shown.

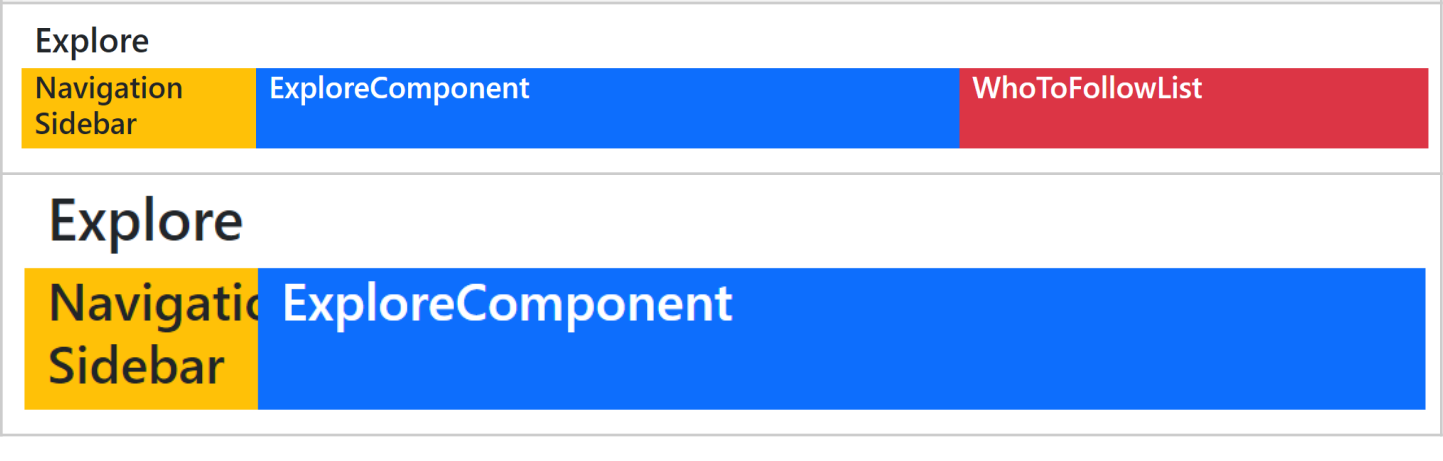
explore/index.js

```
function exploreComponent() {
   $('#wd-explore').append(`
       <h2>Explore</h2>
   `);
}

$(exploreComponent);
```



The explore screen component consists of a row **div** that contains the left, center and right columns, respectively rendering the ***NavigationSidebar***, the ***ExploreComponent***, and ***WhoToFollowList*** components. The code below lays out the screen with just headings in place of the components. Add the code below into ***explore/index.js*** and confirm ***explore/index.html*** renders as shown below. Resize the screen to confirm that the last column appears and disappears based on the size of the window. We've added background and text colors to easily test the layout. Remove these colors when satisfied the screen renders properly. In the sections that follow you'll be asked to implement the three components and replace their respective heading place holders.

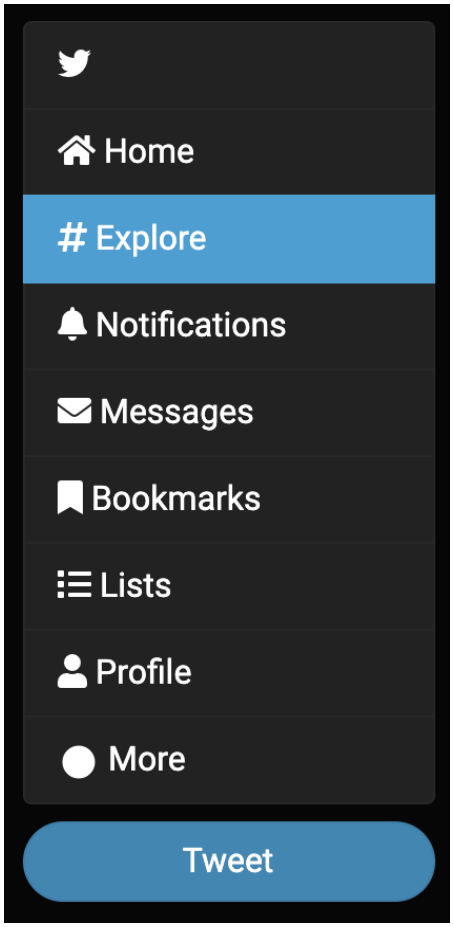
```
function exploreComponent() {
   $('#wd-explore').append(`
       <h2>Explore</h2>
  <div class="row mt-2">
   <div class="col-2 col-md-2 col-lg-1 col-xl-2 bg-warning">
    <h3>NavigationSidebar</h3>
   </div>
   <div class="col-10 col-lg-7 col-xl-6 bg-primary">
    <h3>ExploreComponent</h3>
   </div>
   <div class="d-none d-sm-none d-md-none d-lg-block col-lg-4 col-xl-4 bg-danger">
    <h3>WhoToFollowList </h3>
   </div>
  </div>
   `);
}
$(exploreComponent);
```



In the next sections you will be asked to implement the left navigation side bar, the explore component in the center, and the who to follow list side bar on the right. The components will then be used to build the explore screen. Each component should be able to be tested independently by creating an html document that loads all the necessary CSS and JavaScript libraries. Once you have completed and tested each of the components independently, they can be imported into the explore screen to render the entire page.

3.4 Implement the NavigationSideBar component

Under the ***tuiter*** directory create a directory called ***NavigationSidebar*** and an ***index.js*** file that will implement the component. In the ***index.js*** file, create a function also called ***NavigationSidebar*** that returns the navigation sidebar rendered here on the right. This is the same navigation sidebar you implemented in previous assignments so feel free to reuse the HTML and CSS from that assignment. Below is an example of what the function might look like. To test the component on its own, create ***NavigationSidebar/index.html*** that links all necessary bootstrap stylesheets, the jQuery JavaScript library and the ***NavigationSidebar/index.js*** file. Load the ***index.html*** file and confirm the sidebar renders as shown here on the right. You might need to remove your own styling and comment out some of the other components. The ***index.html*** file is not required, but it can help test the sidebar independently.




```
const NavigationSidebar = () => {
 return(`
   <div class="list-group">
     <a class="list-group-item" href="/">
       <i class="fab fa-twitter"></i></a>
       <!-- continue rest of list, e.g.,
            Home, Explore, Notifications,  Messages, etc. -->
   </div>
   <div class="d-grid mt-2">
     <a href="tweet.html"
        class="btn btn-primary btn-block rounded-pill">
        Tweet</a>
   </div>
 `);
}
export default NavigationSidebar;
```

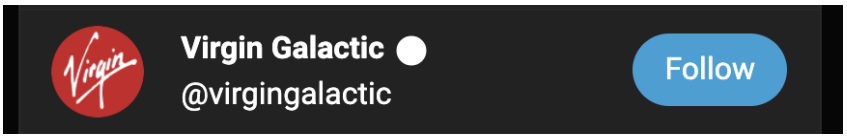
Add the ***NavigationComponent*** to the explore screen as shown below

```
import NavigationSidebar from "./NavigationSidebar/index.js";

function exploreComponent() {
   $('#wd-explore').append(`
       <h2>Explore</h2>
  <div class="row mt-2">
   <div class="col-2 col-md-2 col-lg-1 col-xl-2">
   <!--<h3>Navigation Sidebar</h3>-->
    ${NavigationSidebar()}
   </div>
   <div class="col-10 col-lg-7 col-xl-6 bg-primary text-white">
    <h3>ExploreComponent</h3>
   </div>
   <div class="d-none d-sm-none d-md-none d-lg-block col-lg-4
col-xl-4 bg-danger text-white">
    <h3>WhoToFollowList </h3>
   </div>
  </div>
   `);
}
$(exploreComponent);
```

```
// you might want to comment
// some of these until you complete
// them so you can test
// each component independently



// you might want to comment
// some of these until you complete
// them so you can test
// each component independently
```

3.5 Implement the WhoToFollowListItem component

Now let's focus on the right sidebar. Under the ***tuiter*** directory, in a new folder called ***WhoToFollowList***, create a JavaScript file called ***WhoToFollowListItem.js*** that implements a function called ***WhoToFollowListItem*** that accepts a parameter called ***who***. The parameter represents a single user we might want to follow. The function should return an HTML string that renders as shown here on the right. The data in the ***who*** parameter contains properties that describe each user such as ***avatarIcon***, ***userName***, and ***handle***. Feel free to re-use the HTML you wrote for the previous assignment. To test, you can create an HTML file that links the required bootstrap CSS libraries, jQuery library, as well as any other JavaScript and CSS files. Load the HTML file in your browser and confirm it renders as shown above. An example of the object ***who*** used to render the sample image above is shown below. The HTML file to test your ***WhoToFollowListItem*** component is not required, but is a good idea to be able to test the component independently.



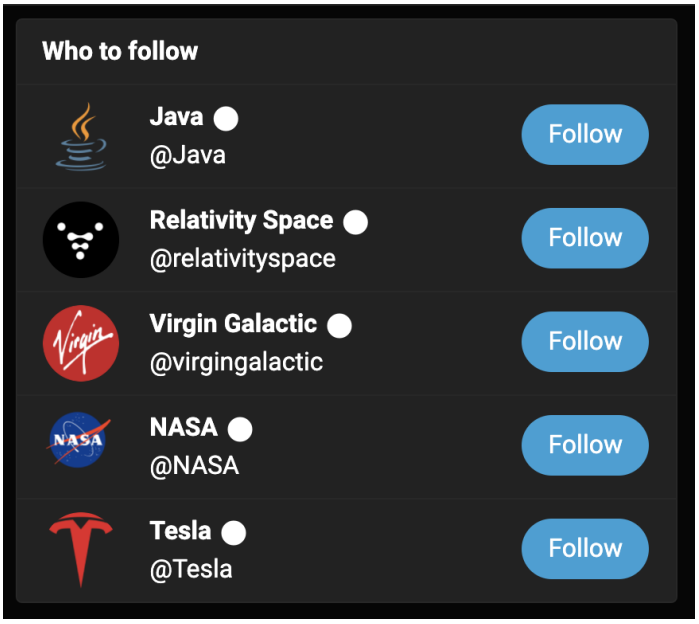

```
{
  avatarIcon: '../../images/virgin.png',
  userName: 'Virgin Galactic',
  handle: 'virgingalactic',
},
```

3.6 Implement the WhoToFollowList component

In the same ***WhoToFollowList*** directory, in an ***index.js*** file, implement a function called the same as the file that renders a list of people as shown here on the right. From ***index.js***, import a file called ***who.js*** that exports the array as shown above. The array declares objects representing the suggested people to follow.

```
export default [
 {   avatarIcon: 'java.png',
   userName: 'Java', handle: 'Java', },
 {   avatarIcon: 'relativity.jpeg',
   userName: 'Relativity Space',
   handle: 'relativityspace', },
 {   avatarIcon: 'virgin.png',
   userName: 'Virgin Galactic',
   handle: 'virgingalactic', },
 {   avatarIcon: 'nasa.png',
   userName: 'NASA', handle: 'NASA', },
 {   avatarIcon: 'tesla.png',
   userName: 'Tesla', handle: 'Tesla', }, ];
```

who.js



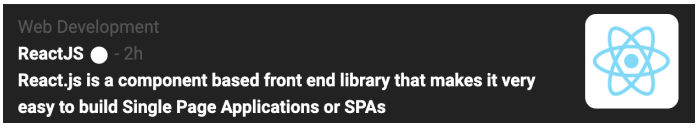
Sample Render

The images in the array refer to image files that should live under ***public/images***. Search for images on the internet similar to the ones shown and save them to the ***images*** folder. Feel free to modify the array or file names as needed. The ***WhoToFollowList*** function in ***index.js*** should import ***WhoToFollowListItem.js*** and ***whos.js*** and iterate over the ***who*** array using the ***WhoToFollowListItem*** function to generate the list shown above. Feel free to re-use the HTML and CSS code you wrote for previous assignments. To test the component

you can create an ***index.html*** file that links all necessary CSS stylesheets and JavaScript libraries and renders the ***WhoToFollowList*** component. Use the file to confirm the who to follow list renders as shown.

```
import WhoToFollowListItem from "./WhoToFollowListItem.js";
import who from "./who.js"
const WhoToFollowList = () => {
   return (`
           <ul class="list-group">
           <!-- continue here -->
           </ul>
`); }
```

3.7 Implement the PostSummaryItem component



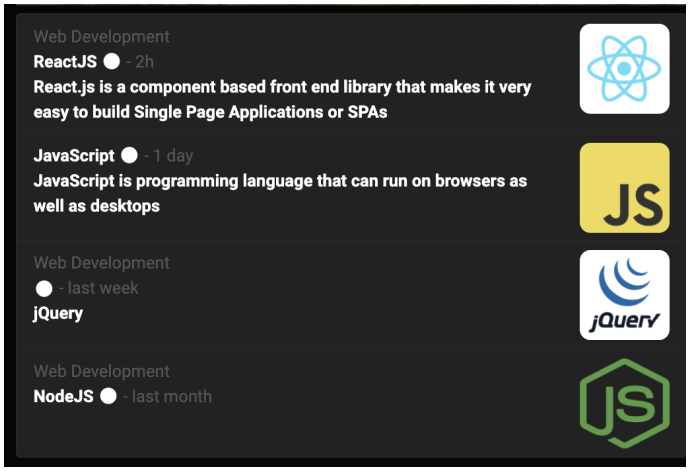
Let's now focus on the center column. In a new directory called ***PostSummaryList***, create a new file called ***PostSummaryItem.js*** that implements a function of the same name. The function should render as shown here on the right. The function should accept an argument called ***post*** that represents a summary of a suggested post. The function should return an HTML string that renders as shown above. The data in the ***post*** contains properties that describe each of the posts such as ***topic***, ***userName***, ***time***, ***title***, and ***image***. An example ***post*** object is shown below.

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

Use the properties to dynamically generate the HTML that renders each of the rows. The function will be used by another function that contains an array of ***post*** instances. That function will iterate over the array and call the ***PostSummaryItem*** function to generate a list of items. Feel free to re-use the HTML and CSS code you wrote for previous assignments. To test ***PostSummaryItem*** independently you can create an HTML document that loads all required CSS and JavaScript libraries and loads the ***PostSummaryItem*** component to confirm it renders correctly. The HTML document is not required, but can help to work on the component independently before reusing it in other components. The ***PostSummaryList*** component implement next will iterate over an array of posts and render each post using the ***PostSummaryItem*** function implemented above.

3.8 Implement the PostSummaryList component

In the same directory, create an ***index.js*** file which implements function ***PostSummaryList*** that renders the list of suggested posts at the bottom of the center column of the explore screen shown here on the right. The function should import an array ***posts*** from a file called ***posts.js*** shown below. The file ***posts.js*** contains objects that represent each of the suggested posts in the list at the bottom of the explore screen. Feel free to re-use the HTML and CSS code you wrote for the previous assignment. Refresh ***render.html*** and confirm that the screen renders as shown at the beginning of the Tuiter section.




```
export default exploreItems = [
   {
       topic: 'Web Development',
       userName: 'ReactJS',
       time: '2h',
        title: 'React.js is a component based front end library that makes it very easy to build
Single Page Applications or SPAs',
       image: '../../images/react-blue.png'
   },
   {
       topic: '',
       userName: 'JavaScript',
       time: '1 day',
       title: 'JavaScript is programming language that can run on browsers as well as desktops',
       image: '../../images/js.png',
       tweets: '123K',
   },
   {
       topic: 'Web Development',
       userName: '',
       title: 'jQuery',
       time: 'last week',
       image: '../../images/jquery.png',
       tweets: '122K',
   },
   {
       topic: 'Web Development',
       userName: 'NodeJS',
       title: '',
       time: 'last month',
       image: '../../images/node.png',
       tweets: '120K',
   },];
```

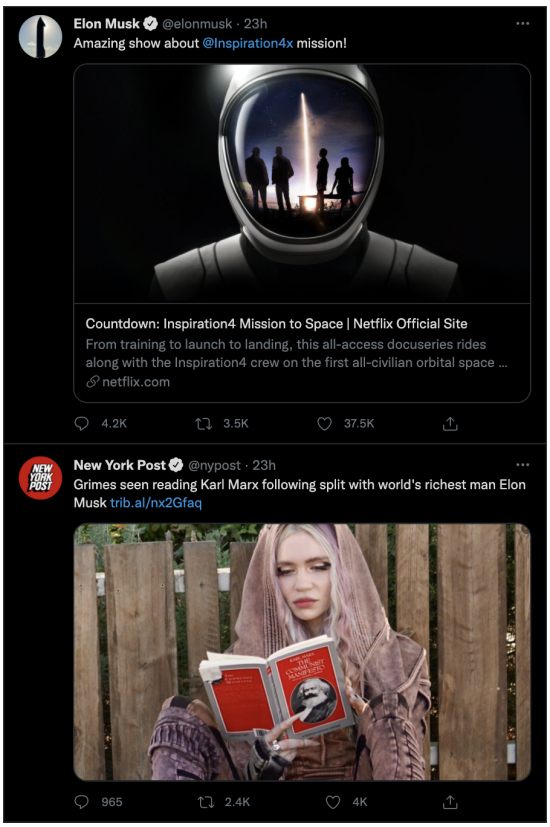
3.9 Implement the ExploreComponent component

Finally, back in the ***ExploreScreen*** folder, create an ***ExploreComponent.js*** file where you'll render the center column of the ***ExploreScreen*** shown here on the right. Import the PostSummaryList component created earlier to render the bottom half of the screen. Feel free to reuse HTML from previous assignments to implement the top half including the search input field, the tabs and the image. Here's an example of how the function might look like

```
import PostSummaryList from "../PostSummaryList";

const ExploreComponent = () => {
   return(`
         <div class="row">
               <!-- search field and cog -->
         </div>
         <ul class="nav mb-2 nav-tabs">
               <!-- tabs -->
         </ul>
         <!-- image with overlaid text -->
         ${PostSummaryList()}
   `);
}
export default ExploreComponent;
```

4 Challenge (required for graduate students)

4.1 Implement a HomeScreen



In a new folder called ***HomeScreen***, use the ***ExploreScreen*** implemented earlier as a guide to implement the home screen shown here on the right. The layout of the screen should be the same as shown. Use the same ***NavigationSideBar*** component used earlier on the left side just like in the ***ExploreScreen*** from earlier. On the right hand side, instead of the ***WhoToFollowList*** component, render the ***PostSummaryList*** component you implemented eralier. In the center column, create a new component called ***PostList*** in a folder of the same name. Use the component ***PostSummaryList*** you created earlier as a guide. Instead of ***PostSummaryListItem***, create a component ***PostItem*** that renders a full post. ***PostItem***s render as shown here on the right showing two instances of posts. Create an array called posts that contains all the information you would need to render the two posts shown here on the right. Use fontawesome icons to render the bottom icons below each of the posts. Make sure the padding and margins are as similar to what is shown in the wireframe.

4.2 Parameterize the NavigationSidebar component

Refactor ***NavigationSidebar*** so it accepts ***active*** as a parameter. If parameter is equal to 'home', then the ***Home*** hyperlink should be highlighted (active), if it's equal to 'explore', then the ***Explore*** hyperlink should be highlighted (active), and so on for all other hyperlinks. Refactor the hyperlinks in ***NavigationSidebar*** so that you can navigate between the ***ExploreScreen*** and ***HomeScreen***. Both are implemented in ***index.html*** files located in folders of the same name as the components. The hyperlink to the ***HomeScreen*** should be ***../HomeScreen/index.html*** and the hyperlink to the ***ExploreScreen*** should be ***../ExploreScreen/index.html***. Make sure at the end of this assignment that these links work as expected allowing navigation between these two screens.

5 Delivery

All your work must be done in a branch called ***a5***. 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 **a5***

```
git  checkout  -b  a5
```

```
# do all your work
```

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

Add, commit and push the new branch

```
git  add  .
git  commit  -am  "a5 JavaScript sp22"
git  push
```

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

In Canvas, submit the following

1. The new URL where your ***a5*** branch deployed to on Netlify
2. The link to your new branch in GitHub.