SENG365 Lab 3
Javascript for the Client and Interaction with APIs using React

SENG365

Ben Adams Morgan English

17th February 2023

Purpose of this Lab

This lab begins our look into client-side development using JavaScript. This and the following labs will prepare you for Assignment 2 where you will create a web application for the API created in Assignment 1.

In this lab we will start from the basics of client-side development with plain old JavaScript and HTML, then we will move on to using a framework, in this case the popular React framework created by Facebook/Meta

1 JavaScript for the Front-end

Where HTML and CSS allow web developers to format the contents of a web page, JavaScript allows them to make the page responsive. For example, HTML is used for creating text boxes and creating buttons, and CSS for styling these by changing their size, colour, or other attributes; whereas JavaScript allows for changing context on the page, creating pop-up messages and validating text in text boxes to make sure the required fields have been filled.

JavaScript makes web pages more dynamic by allowing users to interact with web pages, click on elements, and change the pages.

1.1 Exercise 1: JavaScript Form Validation

1. Create a new folder *lab3* and within this another folder *ex1*.
2. Create a new file *index.html* with your *ex1* folder that contains the code in Listing 1.

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Lab 3: Exercise 1</title>
</head>

<body>
    <form method="GET" action="https://www.google.com/search">
        <input type="text" name="q" id="search_string" />
        <input type="submit" value="Search" />
    </form>
</body>

</html>
```

Listing 1: Example HTML form.

The method, action, and name of the text box are specific values the Google homepage uses to interact with it's server.

3. Open the HTML file in your browser and test. The form will query Google with the search string that is entered into the box.
4. In the opening form tag, add an attribute called `onSubmit` that has a value of `return validateForm()`.

```
<form method="GET" action="https://www.google.com/search" onsubmit="return validateForm()">
```

Listing 2: Updated form with `onSubmit`.

5. At the bottom of the `<body>` tag in your HTML add a new `<script>` tag and include the `validateForm` function. This function checks the value of the text field on the form. If the field is not empty `!=""` then the function should return `true`, else provide a popup message and return `false` as shown in Listing 3.

```
<script type="text/javascript">
    function validateForm() {
        var search_string = document.getElementById("search_string").value;
        if (search_string == "") {
            alert("Search string is empty!");
            return false;
        } else {
            return true;
        }
    }
</script>
```

Listing 3: Validate form function.

6. Open your HTML file in the browser and test your form. The form should not submit to the server unless there is a query entered into the text box.

1.2 Debugging Front-end Applications with Google Chrome

In term 1 we discussed how to properly debug a server-side application using WebStorm's built-in debugger. For client-side applications however, we can make use of tools built into browsers. We will specifically focus on Google Chrome's DevTools[1], however most other browsers have comparable tools. Google provides a great video introduction to the Chrome debugger[2]. The video tutorial only focusses on debugging JavaScript code however the webpage covers of the other functionality for understanding network requests, page layout and styling, and storage.

2 JavaScript Frameworks (React)

React is a very popular framework, scoring 2nd in the 2022 Stack Overflow survey for web frameworks and technologies[3]. Due to this popularity there is a large amount of online documentation and tutorials to help learn React, as well as many third-party libraries that work with React that we can use to make our development easier.

"React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called 'components'."[4]

[1]See `https://developer.chrome.com/docs/devtools/overview/`
[2]See `https://www.youtube.com/watch?v=H0XScE08hy8&ab_channel=GoogleChromeDevelopers`
[3]See: `https://survey.stackoverflow.co/2022/#section-most-popular-technologies-web-frameworks-and-technologies`
[4]Taken from `https://reactjs.org/tutorial/tutorial.html#what-is-react`

2.1 Exercise 2: Getting Started with React

To use React, we must import a few scripts like any other JavaScript file. This can be done by downloading the file, using a package manager (such as npm), or by using a CDN[5]. For simplicity we will use a CDN for this lab, however this is not recommended for proper large scale development.

1. Create a new folder *ex2* and create an HTML file *index.html* adding the code from Listing 4.

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>My Application</title>
</head>

<body>
    <div id="root">
    </div>
    <!-- Import the React, React-Dom and Babel libraries from unpkg -->
    <script type="application/javascript" src="https://unpkg.com/react@18.2.0/umd/react.production.min.js"></script>
    <script type="application/javascript"
        src="https://unpkg.com/react-dom@18.2.0/umd/react-dom.production.min.js"></script>
    <script type="application/javascript" src="https://unpkg.com/babel-standalone@6.26.0/babel.js"></script>
    <script type="text/babel">
    </script>
</body>

</html>
```

Listing 4: Basic HTML page importing React from CDN

Now that we have imported the required React and Babel libraries we can start building our app. You may notice the we have an open and close `<script>`, this is where we will be adding the JavaScript code[6].

2. Copy the code from Listing 5 below into your HTML file between the `<script>` tags.

```javascript
const rootElement = document.getElementById('root')

function App(){
    return (
        <h1>Hello World!</h1>
    )
}

ReactDOM.render(<App/>, rootElement)
```

Listing 5: Basic React app JavaScript code

3. Open *index.html* in your browser to test.

What's Happening?:

Here we get the root element of our HTML (by its id), then we create a simple React component `App` which returns an HTML element, in this case a simple `<h1>` tag saying "Hello World!". Finally we use the `ReactDOM.render()`[7] function to render our React component (now in the form of a custom HTML tag `<App/>`) to the aforementioned root element. Importantly this final function works with the DOM[8] (Document Object Model), binding the result from our component to said DOM.

[5]See `https://en.wikipedia.org/wiki/Content_delivery_network`

[6]In the past we could place code in another file an reference it, however to due security issues this feature is blocked by default on browsers, for some more information see: `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors/CORSRequestNotHttp`

[7]See `https://reactjs.org/docs/react-dom.html`

[8]See `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction`

2.2 Exercise 3: React Functional Components

The example we looked at above is very simple and shows the basics of how we can use React components to render HTML content. In this exercise we will build on this by making use of more featured React components.

In this section we will make use of React functional components (a somewhat recent addition to improve on the previous class components). Functional components are called this for the fact that each component is simply a function, though sometimes it can be hard to really see them as just functions.

1. Create a copy of your folder *ex2* as *ex3*.
2. In the `<script>` tags create a new functional component `ShoppingList` with the code from Listing 6.

```
const ShoppingList = () => {
    return (
        <div>
            <h1>My Shopping List</h1>
        </div>
    )
}
```

Listing 6: React `ShoppingList` functional component

3. In the `App` function replace the `<h1>` return with the `<ShoppingList/>` (now as an HTML tag). **Note:** we have wrapped the `<ShoppingList/>` tag in a `<div>` as we must return exactly one element at the highest level. This allow for more future-proofing going forward if we wanted to add any other components.

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

Listing 7: Returning our `ShoppingList` element

4. Now if we open the *index.html* file we should see that the `<h1>` text from our `ShoppingList` component is being displayed on screen.

2.2.1 Exercise 3.1: Conditional Rendering

React conditional rendering can be done in a few ways. What gets rendered is simply what gets returned from the function so we can have different conditional return statements.

1. Within the ShoppingList component make a new variable called visible

```
    let visible = true
```

Listing 8: Declaring the `visible` variable

2. Wrap the return statement within an if statement

```
if (visible)
    return (
        <div>
            <h1>My Shopping List</h1>
        </div>
    )
```

Listing 9: Conditional rendering with basic if statement

3. Open the HTML page in your browser and you should see the same content as before. But once you change visible to false, save the file, and refresh the page it should be completely empty. You can experiment using other if/else statements to return other values.

However we can also conditionally render within our JSX[9] using ternary operators. A ternary operator is similar to and if else statement with special formatting. In the case of JSX we use the format
`{conditional ? result_if_true : result_if_false}`
The conditional will be tested and if true the value after the `?` but before the `:` we be returned, and if false the value after the `:`. **Note:** The curly braces surrounding the statement are important, this is how we specify within JSX that we are writing code.

1. Remove the if statement we added above, in its place add the following code

```
return (
    <div>
        {visible ? <h1>My Shopping List</h1> : ""}
    </div>
)
```

Listing 10: Conditional rendering with JSX ternary

2. Refresh the page, and again depending on the value of `visible` our text will (or won't be displayed). Feel free to change what is displayed when `visible` is false by adding some text to the quotation marks or replacing it with some HTML content.

2.2.2 Exercise 3.2: Complex Rendering of Lists

React allows us to render a list (or array) of items with the `Array.map()`[10] function.

1. Create a new variable `shopping_list` that stores a list of items with a name and price.

```
    let shopping_list = [{name:"bread", price: 2.75}, {name: "milk", price: 2.50}, {name: "pasta", price: 1.99}]
```

Listing 11: `shopping_list` declaration

2. HTML has built-in tags for displaying lists so we will make use of these, with `<ul>` being an unordered list. Lets replace the conditional rendering from before.

```
return (
    <div>
        <h1>My Shopping List</h1>
        <ul>
        </ul>
    </div>
)
```

Listing 12: Basic HTML layout with unordered list

3. Now for each item we need an `<li>` (list item) component, but clearly doing this by hand is not the way to go, instead we can call `shopping_list.map()` and for each item return a `<li>` component. **Note:** Make sure to put this code between the `<ul>` tags.

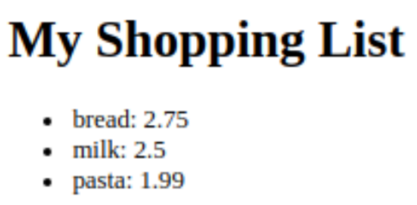
```
                {shopping_list.map((item, index) => (
                    <li key={index}>{item.name}: {item.price}</li>
                    ))}
```

Listing 13: Creating list item components for items in `shopping_list`

4. Now if you refresh your page you should see each item shown with its name and price like so:

[9]JSX will be discussed in more detail in future labs, at this stage simply think of it as a way to mix JavaScript and HTML within one structure. If you want to learn more see: `https://reactjs.org/docs/introducing-jsx.html`

[10]See `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map`


My Shopping List

- bread: 2.75
- milk: 2.5
- pasta: 1.99

2.2.3 Exercise 3.3: React Methods

Finally in this section we will look at making use of methods in our functional component to display computed information.

1. Create a new function `calculate_total` within the `ShoppingList` component and add the code from Listing 14.

```
const calculate_total = () => {
    return shopping_list.map(item => item.price).reduce((prev, next) => prev + next)
}
```

Listing 14: `calculate_total` function

This code sums the price of each item in the `shopping_list` array. For those interested it does this by mapping each element to its price, then using the `Array.reduce()`[11] function to 'walk' through the array element-by-element and add up the values.

2. Now call the function at the bottom of your returned JSX with the following line (but make sure its still within the div tags).

```
                    <h3>Total: ${calculate_total()}</h3>
```

Listing 15: `calculate_total` function

3. Now if you save and refresh your web page you should see the total price displayed on the page.

3 Interacting with APIs Using React

In the previous section we introduced React with some basic examples, from here on we will be putting these into practice in larger scale example along with more complex features. There is not enough time in the labs to teach everything React has to offer so as you work through the remainder of this and the following labs we recommended consulting the React documentation[12].

3.1 Exercise 4: Initial Setup

We are going to build a front end application for the chat app API we created in lab 2. Start by running the API that you created for lab 2 (or downloading the complete version from Learn).

Note: If your API is hosted at a different location to your client application (i.e. a resource has to make a cross-origin HTTP request to a different domain, protocol, or port) then you will need to add CORS[13] support to the API code. This allows your API to accept requests from different 'origins'. One method to add CORS support is to add the code from Listing 16 into the *express.ts* config file. **Note:** This has already been included in the implementation available on Learn.

```
app.use((req, res, next) => {
    res.header("Access-Control-Allow-Origin", "*");
    res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
```

[11]See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce
[12]See https://reactjs.org/docs/getting-started.html
[13]See https://enable-cors.org/

```
    res.header("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
    next();
});
```

Listing 16: Express CORS support

3.2 Exercise 5: Interacting with an API - User Functionality

In this exercise we will create a React web application that calls our API from lab 2, implementing all of the user features. Let's first remind ourselves of API calls we can make regarding users. **Note:** You may prefer to refer to the full API spec provided alongside the lab 2 handout.

URI	Method	Action
/api/users	GET	List all users
/api/users/:id	GET	List a single user
/api/users	POST	Add a new user
/api/users/:id	PUT	Edit an existing user
/api/users/:id	DELETE	Delete a user

Table 1: Basic API specification for user endpoints..

3.2.1 Exercise 5.1: List All Users

We will start by simply getting a list of all users and displaying them on the screen.

1. Create a new directory *ex5*.
2. Create an *index.html* file and add the code from Listing 17 below.

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>My Application</title>
    </head>
    <body>
        <div id="root">
        </div>
        <!-- Import the React, React-Dom and Babel libraries from unpkg -->
        <script crossorigin src="https://unpkg.com/react@18.2.0/umd/react.development.js"></script> <!--Note:
            development versions of react-->
        <script crossorigin src="https://unpkg.com/react-dom@18.2.0/umd/react-dom.development.js"></script>
        <script type="application/javascript" src="https://unpkg.com/babel-standalone@6.26.0/babel.js"></script>
        <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
        <script type="text/babel">
        </script>
    </body>
</html>
```

Listing 17: `index.html` page

3. Add the code from listing 18 below into the `<script>` tags of the HTML file.

```javascript
const rootElement = document.getElementById('root')

const UserList = () => {

}

function App() {
    return (
        <div>
```

```
            <UserList />
        </div>
    )
}

ReactDOM.render(
    <App />, rootElement
)
```

Listing 18: Boilerplate React code for users.

4. Let's add a variable `users` and a callback `setUsers` at the top of our component that manages the state of a list of users.

```
    const [users, setUsers] = React.useState([])
```

Listing 19: Defining `users` state with React

5. Add the `useEffect` hook and the `getUsers` method to your component. This will request the users from the API when the page loads.

```
    React.useEffect(() => {
        const getUsers = () => {
            axios.get('http://localhost:3000/api/users')
                .then((response) => {
                    console.log(response.data)
                    setUsers(response.data)
                }, (error) => {
                    console.log(error)
                })
        }
        getUsers()
    }, []) // empty array so effect only runs once
```

Listing 20: `getUsers` `useEffect` hook

Note: The second argument for a `useEffect` hook is the dependency array. This is used to specify what values the hook is dependent on, such that if they change the hook needs to be run again. For a simple page load hook we can simply leave it empty.

6. Create a function `list_of_users` that creates a list item for each user and displays their username.

```
    const list_of_users = () => {
        // Work around for map not working on state lists in render, here we explicitly call it when ever render is
             called
        return users.map((item) =>
            <li key={item.user_id}>
                <p>
                    {item.username}
                    <button onClick={() => deleteUser(item)}>Delete</button>
                </p>
            </li>)
    }
```

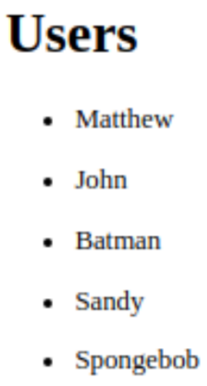
Listing 21: `list_of_users` function to display users as a list

7. Finally return the following from your `UserList` component.

```
    return (
        <div>
            <h1>Users</h1>
            <ul>
                {list_of_users()}
            </ul>
        </div>
    )
```

Listing 22: HTML content being returned from the component

8. Open *index.html* in your browser and test that the users are displayed like so:



Users

- Matthew
- John
- Batman
- Sandy
- Spongebob

Note: If no users show up, make sure to check that you have some in your database. If it still doesn't seem to be working check Chrome's DevTools to see if any errors are being printed to the browser console.

3.2.2 Exercise 5.2: Adding a New User

Next we want to write the functionality that allows us to add a new user.

1. In your JS code add another state variable and callback `username` and `setUsername`.

```
const [username, setUsername] = React.useState("")
```

Listing 23: Defining `username` state with React

2. Add the following `addUser` method to the component.

```
const addUser = () => {
    if (username === "") {
        alert("Please enter a username!")
    } else {
        axios.post('http://localhost:3000/api/users', { "username": username })
    }
}
```

Listing 24: `addUser` function

3. In your return section add the following HTML elements, these will add a text box and button to the page.

```
        <h2>Add a new user:</h2>
        <form onSubmit={addUser}>
            <input type="text" value={username} onChange={updateUsernameState} />
            <input type="submit" value="Submit" />
        </form>
```

Listing 25: Add user html form

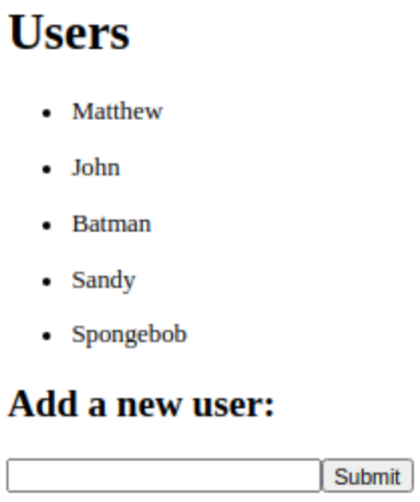
4. You may notice here that we have an `onChange` parameter for our text input that references a method. Let's add that method. **Note:** This handles the updating of our username state as the user types into the text box.

```
const updateUsernameState = (event) => {
    setUsername(event.target.value)
}
```

Listing 26: `updateUsernameState` helper function

5. Finally refresh your page and test that the text box and button are added to the page like so. Test that when you add a user they are automatically added to the list above. **Note:** In this case a full re-render will be done,

re-fetching the users from our api, this happens because we have submitted a form. Whilst this is easy enough to avoid by returning false from our `onSubmit` method, it helps us here so we will leave it as it.



3.2.3 Exercise 5.3: Deleting a User

1. Add the following method to your component. **Note:** On the successful deletion of the user from the database using the API, we loop through out list of users to remove that same user. This allows us to remove the user from the DOM without having to refresh the page (which doesn't happen automatically in this example since we aren't submitting a form).

```
const deleteUser = (user) => {
    axios.delete('http://localhost:3000/api/users/' + user.user_id)
        .then((response) => {
            setUsers(users.filter(u => u.user_id != user.user_id))
        })
}
```

Listing 27: `deleteUser` function

2. In your `list_of_users` function add the following button within the `<p>` tag after the username

```
                <button onClick={() => deleteUser(item)}>Delete</button>
```

Listing 28: `deleteUser` function

3. Open in your browser and test that delete buttons are added next to each username, and that clicking the delete button deletes the related user.

3.2.4 Exercise 5.4: Editing a User

Using the code examples from the previous exercises and the `axios.put()`[14] method, implement the code to edit a user. Don't worry about imperfections as next week's lab will look at how to properly structure your React applications, and we will be making our way back to TypeScript.

3.3 Exercise 6: Implement the Rest of the Application - Optional

This last exercise is optional, carry on working until you are comfortable with the concepts covered.

Using the details and API specification given in lab 2, create the rest of the chat application. You may want to implement the different aspects of the application in different pages. This is also a good chance to practice styling and responsiveness using HTML and CSS, refer to the HTML/CSS pre-lab for more information about these concepts.

[14] See `https://axios-http.com/docs/api_intro`