Lab 3: Javascript for the client and interaction with APIs using React

1 JavaScript for the front end

Taken from: http://study.com/academy/lesson/what-is-javascript-function-uses-quiz.html

Where HTML and CSS allow web developers to format the content of a web page, JavaScript allows them to make the page dynamic. For example, HTML/CSS allows for making text bold, creating text boxes, and creating buttons, whereas JavaScript allows for changing text on the page, creating pop-up messages, and validating text in text boxes to make sure 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 lab_4, and within this another folder ex_1
2. Create a new file 'index.html' with your ex_1 folder that contains the following code. Note the method, action and name of the text box. These are the values that the form on the Google homepage uses to interact with its server.

```
<!DOCTYPE html>
<html lang="en">
   <head>
       <meta charset="UTF-8">
       <title>Lab 4: 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>
```

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()">
```

5. Create a new file called 'validate.js'
6. Inside your JavaScript filem write the validateForm() function. This function should check 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

```
function validateForm(){
   var search_string =
document.getElementById("search_string").value;
   if (search_string == "") {
       alert("Search string is empty!");
       return false;
   } else {
       return true;
   }
}
```

7. In your HTML file, use the script tag to import your JavScript file. Read here to learn why we import JavaScript at the bottom of our webpage: https://robertnyman.com/2008/04/23/where-to-include-javascript-files-in-a-document/

```
…
       </form>
       <script type="text/javascript" src="./validate.js"></script>
   </body>
</html>
```

8. Open your HTML file in the browser and test your form. The form will 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, however most other browsers have comparable tools. Watch this video to learn the basics of debugging using Chrome's DevTools, this will likely be a priceless tool when you are working on your second assignment: https://www.youtube.com/watch?v=H0XScE08hy8. This tutorial only focuses on debugging our javascript code, however the dev tools also provide useful functionality for understanding network requests, page layout and styling, and storage.

2 JavaScript Frameworks (React)

“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’.” Taken from: https://reactjs.org/tutorial/tutorial.html#what-is-react.

2.1 Exercise 2: Getting started with React

To use React, a script is imported into your project like any other JavaScript file. This can be done by downloading the file, using a package manager (such as npm; we will look into this in future labs), or by using a CDN (https://en.wikipedia.org/wiki/Content_delivery_network). For simplicity we will use a CDN for this lab (shown below), however this is not recommended for proper large scale development.

1. Create a new folder “ex_2” and an html file called “index.html” within, then add the following code.

```
<!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@16.0.0/umd/react.production.min.js"></
script>
       <script type="application/javascript"
src="https://unpkg.com/react-dom@16.0.0/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" src="./app.js"></script>
   </body>
</html>
```

Now that we have imported the required React and Babel libraries we can start building our app. You may notice the final script here refers to a local file "./app.js", let's create this file and start with a hello world example.

2. Create a new file "app.js" and add the following code

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

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

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

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() function to render our React component (now in the form of a custom HTML tag) to the aforementioned root element. Importantly this final function works with the DOM (Document Object Model), binding the result from our component to said DOM.

Note: For further reading about the DOM refer to: https://developer.mozilla.org/enUS/docs/Web/API/Document_Object_Model/Introduction#What_is_the_DOM, and for more information on the ReactDOM.render method refer here: https://reactjs.org/docs/react-dom.html#:~:text=Note%3A-,ReactDOM.,diffing%20algorithm%20for%20efficient%20updates.

3. Open "index.html" in the browser to test

2.2 Exercise 3: React functional components

The example we looked at above is very simple and simply shows how we can use React components to render some HTML content. In this section 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 recent addition to improve on the previous class components, which you may have experience with if you have worked with React before). 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 functions.

1. Create a copy of the folder used for the example before now as “ex_3”.
2. In the app.js file we will create a new functional component called “ShoppingList” with the following code; for now we will simply return a heading.

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

3. In the App function replace the H1 return with the ShoppingList (now as an HTML tag). **Note:** we have surrounded the ShoppingList tag with div tags as we must return exactly one element at the highest level. This allows for more future-proofing going forward if we wanted to add any other components

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

4. Now if we open the index.html file we should see that the H1 text from our ShoppingList is being displayed on screen.

2.2.1 Exercise 3.1: Conditional rendering

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

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

```
let visible = true
```

2. Wrap the return statement within an if statement

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

```
    )
```

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

However we can also conditionally render within our JSX using ternary operators. A ternary operator is similar to an if else statement with special formatting. In the case of JSX we use: {conditional?result_if_true:result_if_false}

The conditional will be tested and if true the value after the “?” but before the “:” will be returned, and if false the value after the “:”.

Note: You likely noticed that the above example is surrounded in curly braces. This is how we specify within JSX that we are writing code.

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

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

2. Refresh the page and depending on your visible value you should be able to see the H1 content or not. We can also add what we want to display for a false condition in place of the quotation marks.
   Note: Remember as before these expect JSX expressions with exactly one root element

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() function.

1. Create a new variable shoppping_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}]
```

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

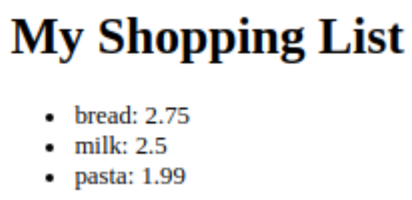
```
return (
   <div>
       <h1>My Shopping List</h1>
       <ul>

       </ul>
   </div>
)
```

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 an <li> component. **Note:** Make sure you put this within the ul tags. (For more

information on the map function refer to: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

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

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



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 following code. **Note:** This code sums the price of each item in our shopping_list array (if you are not comfortable with the code in this form feel free to rewrite it in the form of a for loop or similar)

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

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

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

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

3 Interacting with APIs using React

In the section before this we introduced React with some basic examples, from here on we will be putting these into practice in larger scale examples along with some 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 lab and next week's lab feel welcome to consult the React documentation https://reactjs.org/docs/getting-started.html.

3.1 Exercise 4: Initial setup

We are going to build a front end application for the chat app API that we wrote 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 support to the API code. This allows your API to accept requests from different domains. One method to add CORS support is to add the below code into your ‘**express.ts**’ config file. More information about CORS can be found at: https://enablecors.org/index.html.

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

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. Lets first remind ourselves of the API calls that we can make regarding the users. **Note:** You may prefer to refer to the 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

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 ‘ex_5’
2. Create an index.html file and an app.js file
3. Add the boilerplate code to the HTML file as we did in the earlier examples. Here we will also include the Axios library, which allows us to make http requests.
   https://www.npmjs.com/package/axios

```
<!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@17/umd/react.development.js"></script>
<!--Note: development versions of react-->
       <script crossorigin src="https://unpkg.com/react-
dom@17/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" src="app.js"></script>
   </body>
</html>
```

4. In your app.js file, add the boilerplate code below as we did in earlier examples, this time with a functional component “UserList”

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

const UserList = () => {

}

function App() {
   return(
       <div>
           <UserList/>
       </div>
   )
}

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

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

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

6. Add the following method to your component

```
const getUsers = () =>  {
   axios.get('http://localhost:3000/api/users')
       .then((response) => {
           console.log(response.data)
           setUsers(response.data)
       }, (error) => {
           console.log(error)
       })
}
```

7. Add a new useEffect hook to run the getUsers function when the webpage loads

```
React.useEffect(() => {
   getUsers()
}, []) // empty dependency array so effect only runs once
```

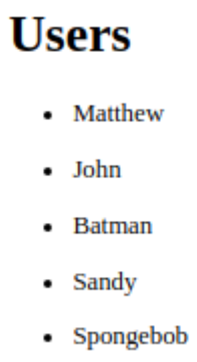
8. Create a function “list_of_users” that creates a list item for each user displaying the username

```
const list_of_users = () => {
   return users.map((item) =>
        <li key={item.user_id}> <p>{item.username}</p> </li>)
}
```

9. Finally return the following from your UserList component

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

10. Run in your browser and test that the users are displayed like so



Users

- Matthew
- John
- Batman
- Sandy
- Spongebob

Note: If no users show up, please check that you have some in your database. If an issue still persists, double check your code and check that there are no issues being printed to your browser console, this can be accessed by pressing F12 on Google Chrome.

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 app.js add another state variable and callback username and setUsername

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

2. Add the following “addUser” method to your component

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

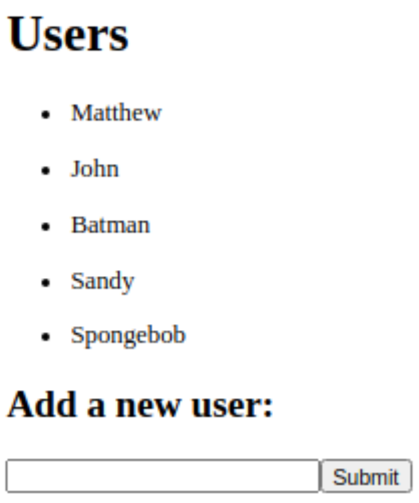
3. In your return section add the following HTML elements, these will add a textbox and button to the page

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

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 textbox

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

5. Finally refresh your page and test that the textbox 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 is because we have submitted a form



Users

- Matthew
- John
- Batman
- Sandy
- Spongebob

Add a new user:

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 our 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 in this example since we are not 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))
       })
}
```

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

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

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

3.2.4 Exercise 5.4: Editing a user

Using the code examples from the previous parts of this exercise and the ‘axios.put()’ method, implement code to edit a particular 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, or contain it all to use just one page. Add styling to your application to make it responsive (using the HTML/CSS pre-lab for help).