SENG365 Lab 4
Structuring Client-side Applications in React

SENG365

Ben Adams Morgan English

28th April 2023

Purpose of this Lab

In last week's lab we looked at using React to create client-side application. In this lab we build on the technologies we applied last time with a focus on structuring our application for maintainability, scalability, and readability.

1 Exercise 1: Initial Setup (Same as Lab 3)

We are again building a front-end application for the chat app API we wrote in Lab 2. For this we will need to run the chat app API as discussed Lab 3, if there any issues please refer back to that Lab.

2 Structuring Large Applications with React

In this lab, we will rewrite the client-side chat application built in last week's lab. The difference being that this week, we will look at structuring our application better, making use of TypeScript, and looking at some other helpful libraries.

2.1 Single Page Applications (SPAs)

Single-Page Applications (SPAs) are Web apps that load a single HTML page and dynamically update that page as the user interacts with the app.

SPAs use AJAX and HTML5 to create fluid and responsive Web apps, without constant page reloads. However, this means much of the work happens on the client side, in JavaScript. [1]

3 Exercise 2: Planning

This exercise is going to run through how to implement the 'User' functionality. The first thing you should do before developing an application is to plan how it will look and how a user will interact with the functionality (and the routes that will hold each page). For the User functionality, let's look at how a user will interact with the application in Figure 1.

[1] Taken from: `https://learn.microsoft.com/en-us/archive/msdn-magazine/2013/november/asp-net-single-page-applications-build-modern-responsive-web-apps-with-asp-net`

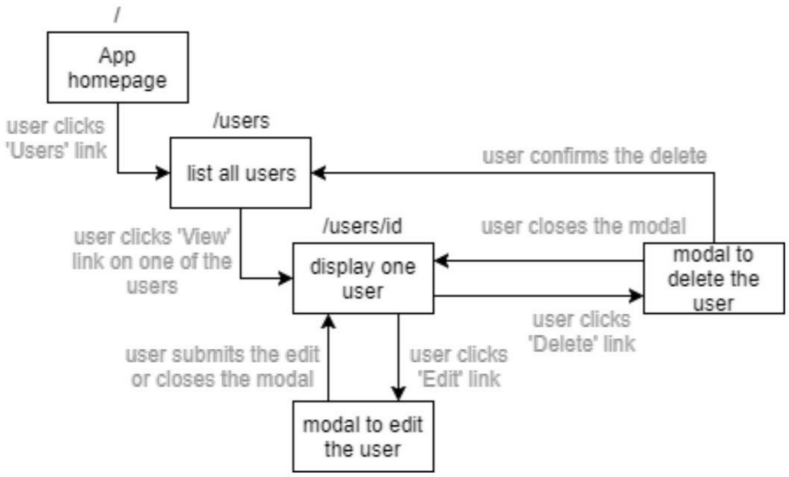

Figure 1: User functionality plan.

We start at the applications home page. The user clicks the 'Users' link to navigate to `/users`. The `/users` page includes a list of all users, each with a link to view that specific user (located at `/users/id`). When viewing an individual user, there is the option to 'Edit' or 'Delete' that user. Editing the user opens a modal with a form to edit the user. If the delete button is clicked, another modal is displayed asking the user to confirm the action. If the user closes the delete modal, they are taken back to `/users/id`. If they confirm the delete operation, then the specified user is deleted and they are sent back to `/users`.

Note: A modal[2] is a pop-up that appears on top of content and provides extra information or the ability to complete some extra action.

1. Plan out the rest of the application, this can just be a rough sketch for now and may change when you get to implementation. However, sketching the application structure out before coding will allow you to get an idea of the 'bigger picture' before you start implementing which can help avoid making small oversights that lead to refactoring later down the line.

4 Exercise 3: create-react-app

create-react-app[3] is a tool that helps you create React projects easily. create-react-app allows for a degree of customisation with the use of templates (in our case TypeScript). create-react-app creates a workflow using **webpack**[4], which among other features allows us to run code on a development server with one command (with real-time updating as we change the code).

1. Create a directory *lab4* and navigate to it in the terminal.
2. Install `create-react-app` using npm[5]with the following command:
   `npm install create-react-app`
3. Run the command:
   `npx create-react-app chat-app --template typescript`
   Note: This may take a few minutes
4. In your *lab4* directory, a new *chat-app* directory should have been created. This contains the skeleton for our project. Have a look around and get familiar with the new structure.
   (a) Inside *chat-app/src* we have the *index.html* file along with *App.tsx*[6] (which replaces our old *app.js* from

[2]See `https://getbootstrap.com/docs/4.0/components/modal/`
[3]See: `https://create-react-app.dev/`
[4]See `https://webpack.js.org/concepts/`
[5]Whilst this install can be done globally, useful if you are creating many React applications, with the `-g` flag it requires administrator privileges so will not work on the lab machines
[6]JSX/TSX is an embeddable XML-like syntax React uses to allow defining both JavaScript/TypeScript and HTML. With the difference between JSX and TSX being for JavaScript and TypeScript respectively. For more information see: `https://www.typescriptlang.org/docs/handbook/jsx.html`

our previous lab)

5. There are a few things we can remove to make our application a little cleaner, though this is not a requirement.
   (a) Remove the *App.text.tsx* and *setupTests.ts* files within the src folder (within the scope of this course we do not expect you to test front-end code).
   (b) Within *index.tsx* in your *src* folder remove the `React.StrictMode` HTML tags and the `reportWebVitals()` method call and import statement. Finally delete the *reportWebVitals.ts* file.
   (c) You can also remove the *node_modules* folder and *package.json* files (along with any others) generated at the root of your *lab4* directory as these were only needed for running the create-react-app command.
6. Create a *.env* file in your *chat-app* directory. Within this add the line:
   `PORT=8080`
   This specifies the port to use when running the application (if you have something running on this port simply choose another one e.g., 8081)
7. Navigate to the *chat-app* directory in your terminal and run:
   `npm run start`
   A new tab should open in your browser with the default React page similar to the image below (if it does not automatically open click the link in the terminal, or navigate to `localhost:8080` in your browser manually).

5 Exercise 4: Routing with react-router-dom

Now that we have our project set up, we need to define a client-side router so that we can create our desired structure. A client-side router works similarly to the server-side express router we used in earlier labs and assignment 1, matching URLs to specific elements.

To implement routing in React we will use the third party routing library 'react-router-dom'[7].

1. In the terminal, navigate to your *chat-app* directory.
2. Run the command:
   `npm install --save react-router-dom`
   Note: the `--save` will save the module to our dependencies in the *package.json* file.
3. In *App.tsx* replace the default code with the following from Listing 1.

```
1  import React from 'react';
2  import './App.css';
3  import {BrowserRouter as Router, Routes, Route} from "react-router-dom";
4  import Users from "./components/Users";
5  import User from "./components/User";
6  import NotFound from "./components/NotFound";
7
8  function App() {
9    return (
10     <div className="App">
11       <Router>
12         <div>
13           <Routes>
```

[7] See `https://v5.reactrouter.com/web/guides/quick-start`

```
            <Route path="/users" element={<Users/>}/>
            <Route path="/users/:id" element={<User/>}/>
            <Route path="*" element={<NotFound/>}/>
          </Routes>
        </div>
      </Router>
    </div>
  );
}

export default App;
```

Listing 1: `App.tsx` content

4. Now we need to create three different elements we will use for our routes, From the code we can see these are `User`, `Users`, `NotFound` all within the `components` directory. Within the `src` directory create the `components` directory.
5. Create a new *.tsx* file for each of *User*, *Users*, and *Not Found*; within them add the code from Listing 2 (swapping `NotFound` for `User` and `Users` to match each file).

```
const NotFound = () => {
    return (<h1>Not Found</h1>)
}

export default NotFound;
```

Listing 2: Component boilerplate content

Note: Commonly tsx file are placed in either a `components` or `pages` directory. With the distinction being a page is often tied to a route and is composed of many components, with components being the tsx that encapsulates smaller functionality. Within the scope of this lab we do not need to worry about this distinction, however when working on your assignment doing so may help you keep everything organised.

6. Finally, we need to create our type definition file and add the `User` type that we will use throughout the lab. Create a folder `types` within the `src` directory and create a new file `users.d.ts` with the type definition from Listing 3.

```
type User = {
    /**
     * User id as defined by the database
     */
    user_id: number,
    /**
     * Users username as entered when created
     */
    username: string
}
```

Listing 3: `User` type

7. Now run your development server with `npm run start` and test that these routes work by typing them into your search bar, e.g. `localhost:8080/users`, `localhost:8080/users/1`, and `localhost:8080/any`.

6 Exercise 5: Adding the Functionality

6.1 Exercise 5.1: Listing all Users

1. In your terminal install **axios**[8] with the command:
   `npm install --save axios`
   Then import it into your *Users.tsx* file.

[8] See `https://axios-http.com/docs/intro`

```
import axios from 'axios';
```

Listing 4: `axios` import

2. Within the *users.tsx* add the following state and function declarations. Also notice that we have introduced the `React.useEffect` hook, in this case it simply fetches all users when the page loads, we discuss hooks more in Section 6.2 below when dealing with a single user.

```
const [users, setUsers] = React.useState < Array < User >> ([])
const [errorFlag, setErrorFlag] = React.useState(false)
const [errorMessage, setErrorMessage] = React.useState("")
React.useEffect(() => {
    getUsers()
}, [])
const getUsers = () => {
    axios.get('http://localhost:3000/api/users')
        .then((response) => {
            setErrorFlag(false)
            setErrorMessage("")
            setUsers(response.data)
        }, (error) => {
            setErrorFlag(true)
            setErrorMessage(error.toString())
        })
}
```

Listing 5: Users state and function declaration

3. Next, we will create a table in our tsx to render the users shown in Listing 6. **Note:** We have also added an error div that will be displayed if there is an issue with fetching the users.

```
if (errorFlag) {
    return (
        <div>
            <h1>Users</h1>
            <div style={{ color: "red" }}>
                {errorMessage}
            </div>
        </div>
    )
} else {
    return (
        <div>
            <h1>Users</h1>
            <table className="table">
                <thead>
                    <tr>
                        <th scope="col">#</th>
                        <th scope="col">username</th>
                        <th scope="col">link</th>
                        <th scope="col">actions</th>
                    </tr>
                </thead>
                <tbody>
                    {list_of_users()}
                </tbody>
            </table>
        </div>
    )
}
```

Listing 6: Users table

4. Above we call a method `list_of_users` to get all the table rows based on our users, the definition is shown in Listing 7.

```
const list_of_users = () => {
    return users.map((item: User) =>
        <tr key={item.user_id}>
            <th scope="row">{item.user_id}</th>
            <td>{item.username}</td>
            <td><Link to={"/users/" + item.user_id}>Go to
                user</Link></td>
            <td>
                <button type="button">Delete</button>
                <button type="button">Edit</button>
            </td>
        </tr>
    )
}
```

Listing 7: `list_of_users` function

5. If your editor has not automatically added the necessary imports you can add the following to the top of your file.

```
import axios from 'axios';
import React from "react";
import {Link} from 'react-router-dom';
```

Listing 8: Imports

6. If your app is still running the changes should be visible in your browser at `localhost:8080:users`. You can test the error div works by stopping you API server temporarily and refreshing the page.

6.2 Exercise 5.2: Viewing One User

1. Within your *user.tsx* file, add the following code to fetch a single user from the API.
   Note: The `useParams` function allows us to access the `id` variable from our path. Also, here we have included the `getUser` function within the `useEffect` hook, in general any value that we rely on must be in the dependency array[9] (this ensures that it re-runs when one of these changes, however this can introduce an infinite render cycle if one of the dependencies is changed by the hook). Hooks are also discussed in more detail in early term 2 lectures.

```
    const {id} = useParams();
    const navigate = useNavigate();
    const [user, setUser] = React.useState<User>({user_id:0, username:""})
    const [errorFlag, setErrorFlag] = React.useState(false)
    const [errorMessage, setErrorMessage] = React.useState("")
    React.useEffect(() => {
        const getUser = () =>  {
            axios.get('http://localhost:3000/api/users/'+id)
                .then((response) => {
                    setErrorFlag(false)
                    setErrorMessage("")
                    setUser(response.data)
                }, (error) => {
                    setErrorFlag(true)
                    setErrorMessage(error.toString())
                })
        }
        getUser()
    }, [id])
```

Listing 9: User state and function declaration

[9] See `https://reactjs.org/docs/hooks-faq.html#is-it-safe-to-omit-functions-from-the-list-of-dependencies`

2. Add the following jsx to display a single user.
   Note: Again, we include buttons for editing and deleting that are not hooked up yet.

```
if (errorFlag) {
    return (
        <div>
            <h1>User</h1>
            <div style={{ color: "red" }}>
                {errorMessage}
            </div>
            <Link to={"/users"}>Back to users</Link>
        </div>
    )
} else {
    return (
        <div>
            <h1>User</h1>
            {user.user_id}: {user.username}
            <Link to={"/users"}>Back to users</Link>
            <button type="button">
                Edit
            </button>
            <button type="button">
                Delete
            </button>
        </div>
    )
}
```

Listing 10: Displaying a single user

3. Here are the imports if needed

```
import axios from 'axios';
import React from "react";
import {Link, useNavigate, useParams } from "react-router-dom";
```

Listing 11: User state and function declaration

4. Now you should be able to click on a link from a user in your table and view them on a new page.

6.3 Exercise 5.3: Deleting a User

When using the internet, you have likely come across modals, these are popups that appear over a page and commonly provide some form of important information or user input. Often these are used for logging in or confirming an action.

To use modals in our application we first need to import Bootstrap[10] into our project. Whilst there is a node package for Bootstrap we will be using a CDN link similar to Lab 3. In the next lab we will look at using proper UI component libraries through npm.

Note: The CDN links may be out of date. Whilst this shouldn't cause any issues, in the unlikely case they do you can get the latest links from the Bootstrap website[11].

1. Replace your *public/index.html* with the following code from Listing 12 or add the links that are not present in your file.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="stylesheet"
```

[10] See `https://getbootstrap.com/`
[11] https://getbootstrap.com/docs/5.1/getting-started/introduction/

```
        href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.min.css"
        integrity="sha384-B0vP5xmATw1+K9KRQjQERJvTumQW0nPEzvF6L/Z6nronJ3oUOFUFpCjEUQouq2+l"
        crossorigin="anonymous">
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <title>Chat App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <script
            src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/js/bootstrap.min.js"
            integrity="sha384-+YQ4JLhjyBLPDQt//I+STsc9iw4uQqACwlvpslubQzn4u2UU2UFM80nGisd026JF"
            crossorigin="anonymous"></script>
  </body>
</html>
```

Listing 12: Basic *index.html* content

2. Now we can create modals. To start we will create one for deleting a user in the *user.tsx* file (the single user page). Add the code from Listing 14 to the file (directly after the delete button we added earlier).

```
                <div className="modal fade" id="deleteUserModal" tabIndex={-1} role="dialog"
                     aria-labelledby="deleteUserModalLabel" aria-hidden="true">
                    <div className="modal-dialog" role="document">
                        <div className="modal-content">
                            <div className="modal-header">
                                <h5 className="modal-title" id="deleteUserModalLabel">Delete User</h5>
                                <button type="button" className="close" data-dismiss="modal" aria-label="Close">
                                    <span aria-hidden="true">&times;</span>
                                </button>
                            </div>
                            <div className="modal-body">
                                Are you sure that you want to delete this user?
                            </div>
                            <div className="modal-footer">
                                <button type="button" className="btn btn-secondary" data-dismiss="modal">
                                    Close
                                </button>
                                <button type="button" className="btn btn-primary" data-dismiss="modal"
                                        onClick={() => deleteUser(user)}>
                                    Delete User
                                </button>
                            </div>
                        </div>
                    </div>
                </div>
```

Listing 13: Delete user modal

3. Update the button so it interacts with the modal. **Note:** We also add some Bootstrap styling to this button using `className="btn btn-primary"`, for other button styles see the Bootstrap documentation[12].

```
                <button type="button" className="btn btn-primary" data-toggle="modal" data-target="#deleteUserModal">
                    Delete
                </button>
```

[12] See https://getbootstrap.com/docs/5.1/components/buttons/. Note that Bootstrap documentation details these as `class="some styling"`, this is a slight inconsistency with JSX and plain HTML since "class" is a keyword in JS we instead use `className`.

Listing 14: Update delete button to work with modal.

4. Add the `deleteUser` function that is referenced in the modal. **Note:** The `navigate` function will take us back to the users page. We also need to define `navigate` using the `useNavigate` function from react-router-dom, the import from this can simply be added to out existing import statement.

```
const navigate = useNavigate();
const [user, setUser] = React.useState<User>({user_id:0, username:""})
const deleteUser = (user: User) => {
    axios.delete('http://localhost:3000/api/users/' + user.user_id)
        .then((response) => {
            navigate('/users')
        }, (error) => {
            setErrorFlag(true)
            setErrorMessage(error.toString())
        })
}
```

Listing 15: Delete user function

5. Now if you run the application, you should be able to delete a user when viewing them on the single user page. **Note:** You may have also noticed that all the styling has changed (hopefully for the better), this is due to Bootstrap's styling.
6. For a challenge see if you can add deleting users to the list of users. This will require finding a way to specify a specific user (**Hint:** Using the buttons already provided with each user you can define a modal for each user in the list, though you will need to make sure they don't open all at once), you may also want to look back at last week's lab to see how to remove an object from a list in place.

6.4 Exercise 5.4: Editing a User

You now have all the knowledge needed to implement the edit functionality. Here are the basic steps you need to take.

1. Create an empty modal.
2. Create the method and any required state variables.
3. Add a form to the modal, implement the edit form similar to the previous lab.
4. Try out the implementation.
5. For a challenge you can try adding both the edit and delete functionality to the list of users page.

6.5 Exercise 5.5: Creating a New User

Omitted from the previous exercises in this lab are the plans for creating a new user. Implement this functionality on the `/users` page, you may want to have a look back at the last lab and implement similar functionality using the new techniques you have learnt. Such as using a modal or simply applying Bootstrap styling to a form on the page.

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

This last exercise is optional, we suggest working through the rest of the chat-app until you are comfortable with the concepts covered.

In the next lab we take a deeper look at some useful libraries and concepts however a good understanding of what is discussed in this lab will be important. That next lab is designed to provide further information on React to help well-designed application for Assignment 2.

Using the details and API specification given in Lab 2, create the rest of the chat application. Use the previous exercises in this lab to help you structure and implement the application. This is a good opportunity to practice styling and responsive design.