Lab 4: Structuring client-side applications in React

In last week's lab we looked at using React to create client applications. This week's lab will look at how to structure React applications to manage scalability and readability.

1 Exercise 1: Initial setup (same as lab 3)

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. If you do not have access to your code, ask a friend for a copy, or the lab tutor for their solution.

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. 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 config file(make sure this is before your route declarations). More information about CORS can be found at: https://enable-cors.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();
});
```

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 in a way that is scalable and usable. Making use of TypeScript, as we did last term, and some helpful libraries.

2.1 Single Page Applications (SPAs)

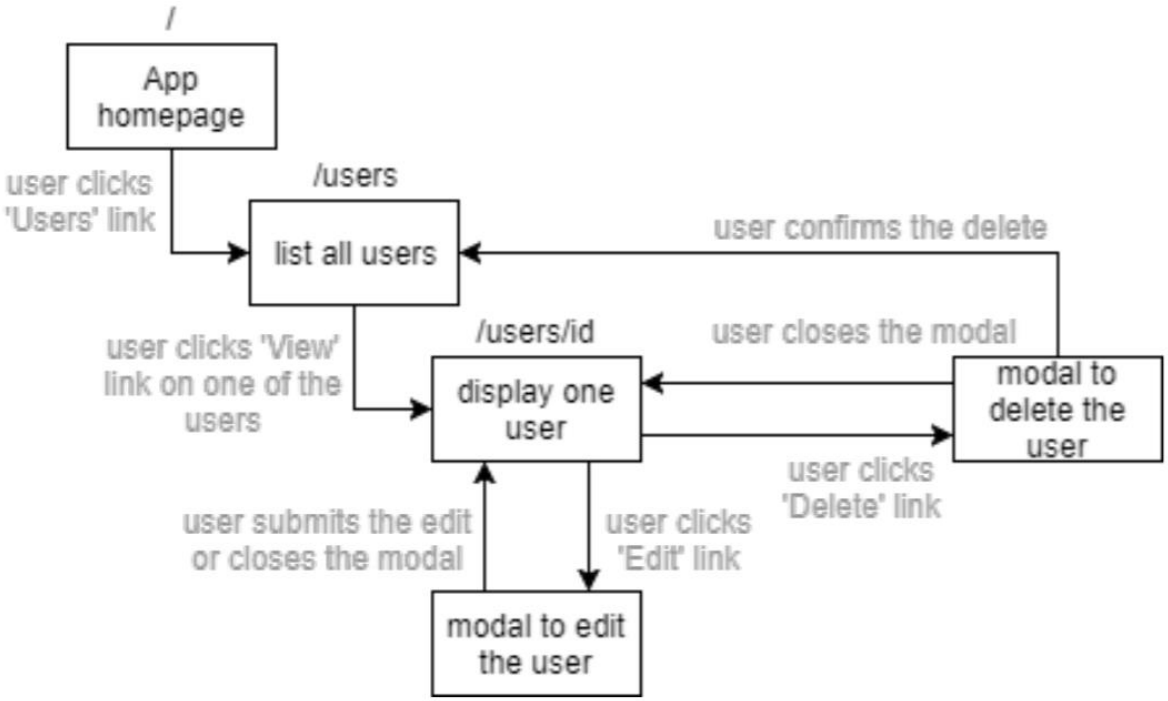
Taken from: https://msdn.microsoft.com/en-gb/magazine/dn463786.aspx
Single-Page Applications (SPAs) are Web apps that load a single HTML page and dynamically update that page as the user interacts with the app.

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.

3 Exercise 2: Planning

This exercise is going to run through how to implement the 'User' functionality. The first thing

you want to do before developing an application is to plan how it will look and how the 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:



Here, we start at the applications homepage. The user clicks the 'Users' link to go navigate to /users. The /users page includes a list of all users, each with a link to view that individual 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, then the user is deleted and the user is sent to /users.

Don't know what a modal is? **Read more:** https://v4-alpha.getbootstrap.com/components/modal/

1. Plan out the rest of the application, this can just be a rough sketch for now and may change when you come to implement. However, sketching the applications structure out before coding will allow you to get an idea of the 'bigger picture' in your head before starting the implementation.

4 Exercise 3: create-react-app

create-react-app is a tool that helps you create a react project 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 (https://webpack.js.org/concepts/), which among other features allows us to run our code on a development server with one command (with real time updates as we change our code).

1. Create a directory called 'Lab 4' and navigate to it in the command line

2. Install create-react-app using npm (whilst this can be done globally with the -g flag, it requires administrator privileges and will not work in the labs) with the following command:
   '*npm install –save 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 Lab 4 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 our index.html file along with App.tsx (which replaces our old app.js from previous labs)
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.test.tsx' and 'setupTests.ts' files with 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 (and related comments). Finally delete the 'reportWebVitals.ts' file.
   c. You can also remove the node_modules folder and package-lock.json (and any other files that were generated at the root of your Lab 5 directory) as these files were only needed for running the 'create-react-app' command and are no longer needed.
6. Create a '.env' file at the root of our "chat-app" folder. Within this add the line "PORT=8080" this specifies the port to use when running the application. (if you have something else running on this port simply choose another one e.g. 8081, this can be helpful especially when working with multiple applications)
7. Navigate to the chat-app directory and run:
   '*npm start*'
   To run the development server. A new tab will open in your browser with the below page (if it does not open automatically click the link in the terminal, or navigate to localhost:8080, or whichever port you specified in the .env file, in your browser). You should see the default react page similar to below

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 have used in earlier labs (and assignment 1). However, client-side routers work using fragment identifiers. In depth knowledge of how they work isn't required for you to implement them, but here's a good video if you'd like to learn more: https://www.youtube.com/watch?v=qvHecQOiu8g

To implement routing in react we will use the third party routing library 'react-router-dom' (https://v5.reactrouter.com/web/guides/quick-start). This exercise will show you how to implement the routing for the chat application.

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

```
import React from 'react';
import './App.css';
import {BrowserRouter as Router, Routes, Route} from "react-router-
dom";
import Users from "./components/Users";
import User from "./components/User";
import NotFound from "./components/NotFound";

function App() {
 return (
   <div className="App">
     <Router>
       <div>
         <Routes>
           <Route path="/users" element={<Users/>}/>
```

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

export default App;
```

4. Now we need to create the three different elements we use for our routes. From the code we can see that these are named ‘User’, ‘Users’, and ‘NotFound’ all within the ‘./components’ directory. Within our src directory create a new ‘components’ directory.
5. Create a new .tsx file for each of **User**, **Users**, and **NotFound**; within them add the code (swapping NotFound for User and Users to match each file). **Note:** Commonly tsx files are placed in either a ‘components’ or ‘pages’ folder. With the distinction being a page is often tied to a route and contains several 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 will help keep everything organised.

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

6. Finally, we should 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 following type definition.

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

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

6 Exercise 5: Adding the functionality

6.1 Exercise 5.1: Listing all users

1. In your command line install axios with the command:
   '*npm install --save axios*'
   and then import this into your Users.tsx file.

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

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 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())
        })
}
```

3. Next, we will create a table in our tsx to render the users. **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>
    )
}
```

4. As you should see we are calling a method list_of_users to get all the table rows based on our users. Let's add that function. **Note:** We have added buttons for edit and delete but at this stage they don't do anything.

```
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>
    )
}
```

5. If your editor has not automatically added the necessary imports please add them to the top of your file.

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

6. If your app is still running the changes should be visible in your browser at localhost:8080/users. Otherwise you may need to re-run the 'npm run start' command. You can test the error div works by stopping your API server temporarily.

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 our getUser function within our useEffect hook, in general any value that we rely on must be in the dependency array (this ensure that it reruns when one of these changes, however this can introduce an infinite render cycle where one of the dependencies is changed by the hook. For a much more in-depth look at what is going on refer to: https://reactjs.org/docs/hooks-faq.html#is-it-safe-to-omit-functions-from-the-list-of-dependencies. Hooks are also discussed in more detail in early term 2 lectures.

```
const {id} = useParams();
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])
```

2. And 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>
    )
}
```

3. Here are the imports if needed

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

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 use input. Often these are used for logging in or confirming an action.

To use modals in our application we first need to import Bootstrap into our project. Whilst there is a node package for Bootstrap we will be using a CDN link similar to Lab 4. 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 please get the latest links from the Bootstrap website: https://getbootstrap.com/docs/5.1/getting-started/introduction/

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

```
<!DOCTYPE html>
<html lang="en">
 <head>
   <meta charset="utf-8" />
   <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min
.css" rel="stylesheet" integrity="sha384-
1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
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@5.1.3/dist/js/bootstrap.bundl
e.min.js" integrity="sha384-
ka7Sk0Gln4gmtz2MlQnikT1wXgYsOg+OMhuP+IlRH9sENBO0LRn5q+8nbTov4+1p"
crossorigin="anonymous"></script>
 </body>
</html>
```

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 following modal code 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">
       <div className="modal-content">
           <div className="modal-header">
               <h5 className="modal-title"
id="deleteUserModalLabel">Delete User</h5>
               <button type="button" className="btn-close" data-bs-
dismiss="modal" aria-label="Close">
               </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-bs-dismiss="modal">
                    Close
                </button>
                <button type="button" className="btn btn-primary"
data-bs-dismiss="modal"
                        onClick={() => deleteUser(user)}>
                    Delete User
                </button>
            </div>
        </div>
    </div>
</div>
```

3. Update the button so it interacts with the modal. **Note:** we also add some Bootstrap styling to this button through the className="btn btn-primary", for other button styles take a look at Bootstrap documentation here: https://getbootstrap.com/docs/5.1/components/buttons/
(you may notice that the Bootstrap documentation details these as class="some styling", this is a slight inconsistency with jsx and plain html, so simply replace "class" with "className")

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

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 our existing import statement)

```
const navigate = useNavigate();
...
const deleteUser = (user: User) => {
    axios.delete('http://localhost:3000/api/users/' + user.user_id)
        .then((response) => {
            navigate('/users')
        }, (error) => {
            setErrorFlag(true)
            setErrorMessage(error.toString())
        })
}
```

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 all open at once), you may also want to look back at last weeks 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 you need to implement the edit functionality. Here are the basic steps that 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. Test
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 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 do provide 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 implement a 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 (using the HTML/CSS pre-lab for help).