Lab 5: More information about React

In the last two labs, we have looked at the basics of React and implemented a small chat application web client project making use of React and our server-side code from last term's labs. This lab will provide some extra information and choices that you may use to help improve your React applications and make them easier to create and maintain.

In this lab we will re-create the chat-app from Lab 4 using a proper UI component library Material UI, more advanced component rendering using props, and state management using Zustand.

This lab does cover some more advanced concepts that can be difficult to easily show in the form of a lab handout. Thus, to get a full understanding of the concepts covered we suggest you make use of online documentation where applicable and play around with the code given to properly understand what is happening.

1 Exercise 1: Initial setup

We are going to refactor our front-end application for the chat-app we wrote in Lab 4, to prepare for this lab we suggest creating a copy of your Lab 4 directory for use. **Note:** To save time don't copy the node_modules folder, instead copy the other files and rerun 'npm install'.

2 Component UI Libraries

When creating any software application, it is prudent to see what libraries are already available, as they provide a structured way to get functionality that may otherwise be very time consuming to create from scratch. We have already seen this to a degree when we used Bootstrap to style our application in Lab 5. Installing a full library through our package manager provides a nicer way to use functionality. In this case we will be looking at component UI libraries as this is an important part of making your website look good.

This lab will specifically look at the Component Library MUI (https://mui.com/). However, there are many different ones that you can use, and due to the popularity of React many of these have great documentation and tutorials you can use. When looking for a UI Library there are several things to keep in mind:

- Is it compatible with my version of React?
- Is there good documentation and examples online?
- Does it look good / Do I like the look of it?

2.1 Material UI (MUI)

2.1.1 Exercise 2.1: Adding Material UI to your project

Material UI requires several packages depending on what you are using. The most important being '`@mui/material`', we will also be making use of icons from '`@mui/icons-material`', and we will need to include '`@emotion/react`' and '`@emotion/styled`'

1. Run the command '*npm install --save <package_name>*' for each of the highlighted packages above. **Or** to save time simply run the shorthand command *'npm i -S @mui/material @mui/icons/material @emotion/react @emotion/styled'*
2. If you have the Bootstrap CDN link included from the previous lab it is ideal to remove it, so you see exactly what styling MUI is producing.
3. Run your app to make sure nothing has broken, then we are ready to start making use of some new components

2.1.2 Exercise 2.2: Replacing modals with Dialogs

Material UI has the component Dialog that acts similar to the modals we made use of in the previous lab. Some differences are the fact that Dialogs depend on our React state and make use of proper HTML tags (including for content within). For more information about the dialog component refer to https://mui.com/components/dialogs/.

1. In lab 4 we suggested creating two modals for each user in the list as an easy way to get around some of the limitations raw modals have. Now with proper Dialogs we can more easily interact with our state. To start with, remove the code for the delete modal.
2. After the table of users place the following code for the delete dialog. There is alot going on here but notice the open={...} this is where we reference our state variable (a boolean) that defines whether the dialog should be shown or not. We also have 'handleDeleteDialogClose' being called on close (this is a method that will handle updating the aforementioned variable, along with any other code we want to run when this happens). Finally note that we have made use of some other new tags we have several inner Dialog tags which are hopefully self-descriptive, we also have new *Button* tags these are explicitly MUI button components.

```
<Dialog
   open={openDeleteDialog}
   onClose={handleDeleteDialogClose}
   aria-labelledby="alert-dialog-title"
   aria-describedby="alert-dialog-description">
   <DialogTitle id="alert-dialog-title">
       {"Delete User?"}
   </DialogTitle>
   <DialogContent>
       <DialogContentText id="alert-dialog-description">
           Are you sure you want to delete this user?
       </DialogContentText>
   </DialogContent>
   <DialogActions>
       <Button onClick={handleDeleteDialogClose}>Cancel</Button>
```

```
        <Button variant="outlined" color="error" onClick={() =>
{deleteUser()}} autoFocus>
            Delete
        </Button>
    </DialogActions>
</Dialog>
```

3. Let's add the related code to interact with the dialog

```
const [openDeleteDialog, setOpenDeleteDialog] =
React.useState(false)
const [dialogUser, setDialogUser] =
React.useState<User>({username:"", user_id:-1})
const handleDeleteDialogOpen = (user:User) => {
    setDialogUser(user)
    setOpenDeleteDialog(true);
};
const handleDeleteDialogClose = () => {
    setDialogUser({username:"", user_id:-1})
    setOpenDeleteDialog(false);
};
```

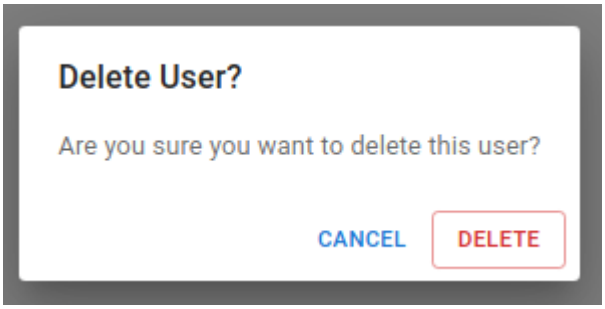
4. Finally update the delete button each user element in the list so it calls the handleDeleteDialogOpen.

```
<Button variant="outlined" endIcon={<DeleteIcon/>} onClick={() =>
{handleDeleteDialogOpen(item)}}>
    Delete
</Button>
```

5. Here are the imports statements to add at the top of your file

```
import {Button, Dialog, DialogActions, DialogContent,
DialogContentText, DialogTitle, TextField} from "@mui/material";
import DeleteIcon from "@mui/icons-material/Delete";
```

6. Run your application and try to delete a user, hopefully you see a nicely styled dialog like so



7. Now use the delete dialog as an example and update the edit modal to use MUI components. Here are a few MUI components you may want to use

```
<TextField id="outlined-basic" label="Username" variant="outlined"
value={usernameEdit} onChange={updateUsernameEditState} />
...
import EditIcon from "@mui/icons-material/Edit";
<EditIcon/>
```

2.1.3 Exercise 2.3: Updating our table

Material UI provides us with a very powerful table component. Within the scope of this lab we will not go into too much depth as it can be quite complex, though there are many examples in the documentation https://mui.com/components/tables/.

1. Replace the old table code with the following

```
<Paper elevation={3} style={card}>
   <h1>Users</h1>
   <TableContainer component={Paper}>
       <Table>
           <TableHead>
               <TableRow>
                   {headCells.map((headCell) => (
                       <TableCell
                           key={headCell.id}
                           align={headCell.numeric ? 'right' :
'left'}
                           padding={'normal'}>
                           {headCell.label}
                       </TableCell>
                       ))}
               </TableRow>
           </TableHead>
           <TableBody>
               {user_rows()}
           </TableBody>
       </Table>
   </TableContainer>
</Paper>
```

2. On the Paper component we specified the style ‘card’, this is a custom style we can add to our code. In this case it simply adds padding and margin so our page doesn’t look too cramped

```
import CSS from 'csstype';
...
const card: CSS.Properties = {
   padding: "10px",
   margin: "20px",
}
```

3. Having a useful table requires us to tell it what datatype it will be displaying. This is particularly important when doing more advanced features such as sorting.

```
interface HeadCell {
   id: string;
   label: string;
   numeric: boolean;
}
const headCells: readonly HeadCell[] = [
   { id: 'ID', label: 'id', numeric: true },
   { id: 'username', label: 'Username', numeric: false },
   { id: 'link', label: 'Link', numeric: false },
   { id: 'actions', label: 'Actions', numeric: false }
];
```

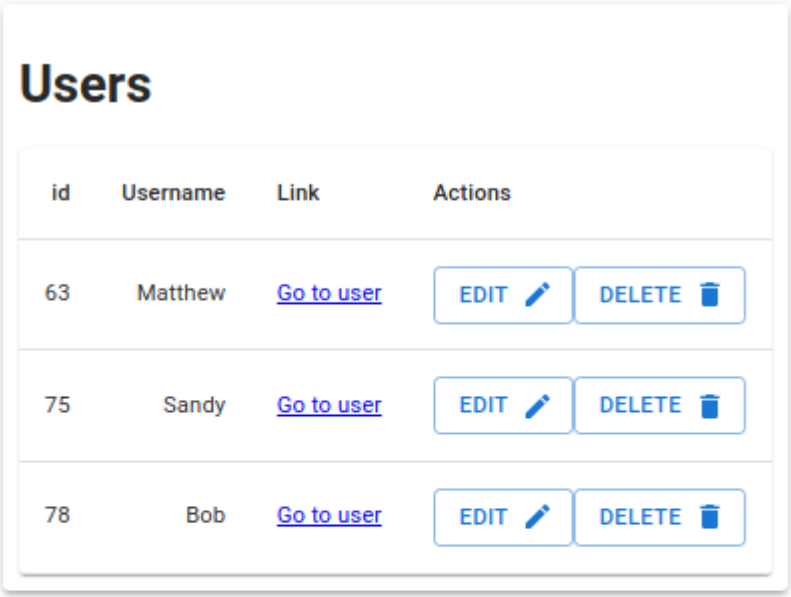
4. Now we need to update our user_rows function so it returns components that can be displayed by the MUI Table component

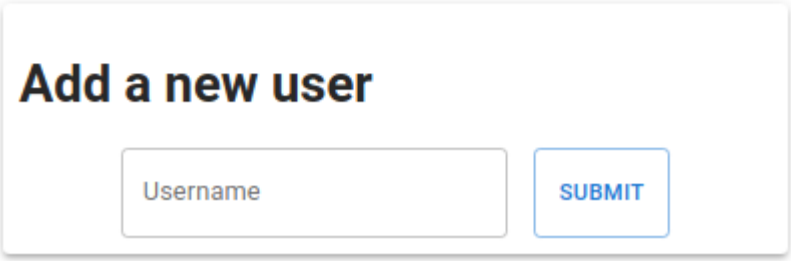
```
const user_rows = () => {
   return users.map((row: User) =>
       <TableRow hover
                 tabIndex={-1}
                 key={row.user_id}>
           <TableCell>
               {row.user_id}
           </TableCell>
           <TableCell align="right">{row.username}</TableCell>
           <TableCell align="right"><Link
to={"/users/"+row.user_id}>Go to user</Link></TableCell>
           <TableCell align="right">
               <Button variant="outlined" endIcon={<EditIcon/>}
onClick={() => {handleEditDialogOpen(row)}}>
                   Edit
               </Button>
               <Button variant="outlined" endIcon={<DeleteIcon/>}
onClick={() => {handleDeleteDialogOpen(row)}}>
                   Delete
               </Button>
           </TableCell>
       </TableRow>
   )
}
```

5. Make sure to include/update the additional imports

```
import EditIcon from "@mui/icons-material/Edit";
import {Button, Dialog, DialogActions, DialogContent,
DialogContentText, DialogTitle, TextField, Paper, Table, TableBody,
TableContainer, TableRow, TableCell, TableHead} from
"@mui/material";
```

6. Now when we open our users page we should see a table similar to below. **Note:** The raised effect comes from the Paper component





7. You may notice that your add user section does not look like above as we have not updated it. Feel free to add the following code to update it. **Note:** this will only work if the add user functionality was completed in the last lab, otherwise it is good practice to implement it now. **Note:** you will have to include the Stack import from @material/ui

```
<Paper elevation={3} style={card}>
   <h1>Add a new user</h1>
   <Stack direction="row" spacing={2} justifyContent="center">
       <TextField id="outlined-basic" label="Username"
variant="outlined" value={username} onChange={updateUsernameState}
/>
       <Button variant="outlined" onClick={() => {addUser()}}>
           Submit
       </Button>
   </Stack>
</Paper>
```

2.1.4 Exercise 2.4: Proper feedback with Alerts and Snackbars

In the last lab we looked at displaying a simple error message in red text when one of our requests failed. Now that we are updating the rest of our application we can also update these. We will cover both the Alert (https://mui.com/components/alert/) and Snackbar (https://mui.com/components/snackbars/) components, though they are somewhat intertwined.

1. Let's replace our old div styled with red colour with a proper error Alert. Remove the old if statement and add the following code to the main div. using && we can do simple conditional rendering.

```
{errorFlag && <Alert severity="error">
   <AlertTitle>Error</AlertTitle>
   {errorMessage}
</Alert>}
```

2. We can also wrap an alert in a SnackBar component. A SnackBar displays messages that popup somewhere on the screen normally from one of the corners to inform the user of something, often fading over time. For this example, we will create a success SnackBar that pops ups whenever we edit a user successfully. Add the following code below your Dialogs.

```
<Snackbar
   autoHideDuration={6000}
   open={snackOpen}
   onClose={handleSnackClose}
   key={snackMessage}
>
   <Alert onClose={handleSnackClose} severity="success" sx={{
width: '100%' }}>
        {snackMessage}
   </Alert>
</Snackbar>
```

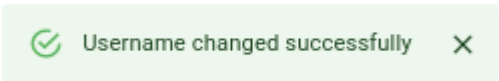
3. Similar to a Dialog we have a specific state boolean that keeps track of whether the snackbar is shown or not. Along with a handler to close it, in this case we specifically need to check that it is not a 'clickaway' event, otherwise any click would close the snackbar.

```
const [snackOpen, setSnackOpen] = React.useState(false)
const [snackMessage, setSnackMessage] = React.useState("")
const handleSnackClose = (event?: React.SyntheticEvent | Event,
reason?: string) => {
   if (reason === 'clickaway') {
       return;
   }
   setSnackOpen(false);
};
```

4. Finally, within our editUser function set the snack message and visibility

```
setSnackMessage("Username changed successfully")
setSnackOpen(true)
```

5. Try and edit a user and check that the alert appears in the bottom left corner like so



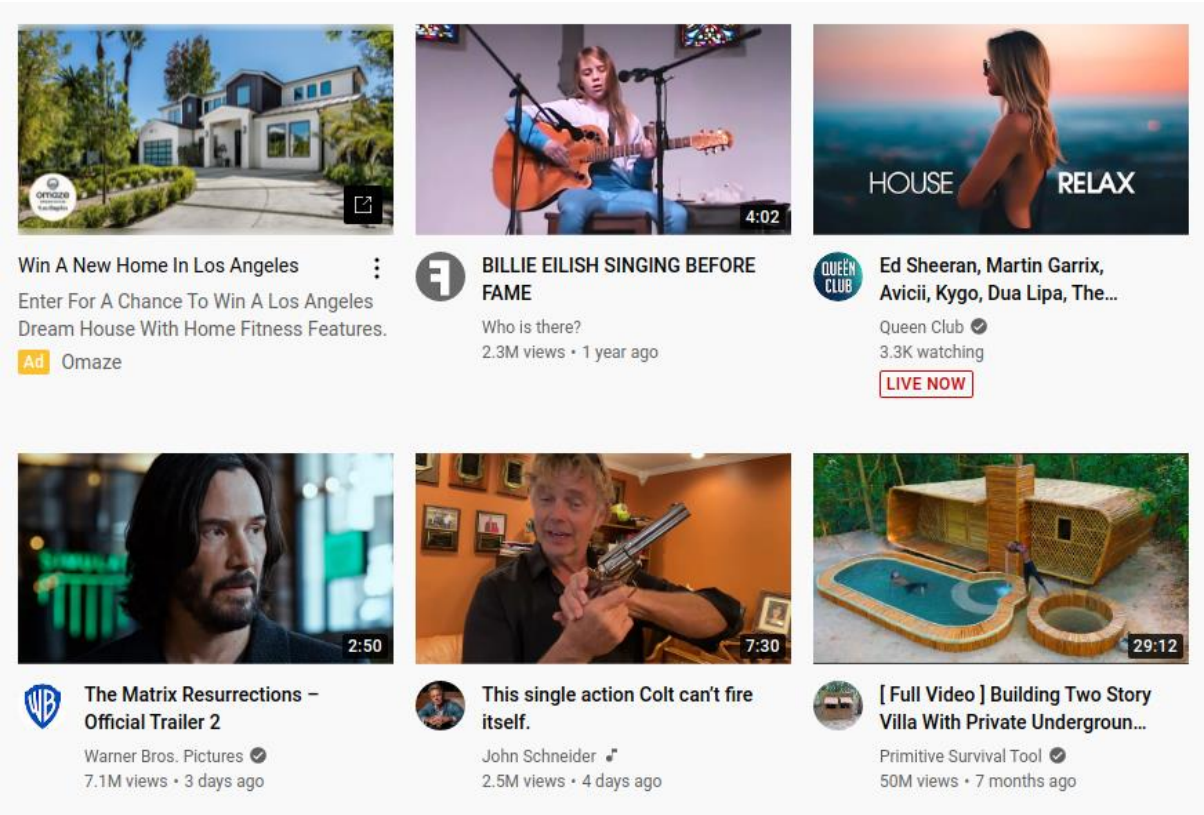
6. Optionally add the same code from step 4 to show a confirmation message for deleting and adding a user

Note: The SnackBar will appear in the bottom left corner by default but this can easily be changed, refer to the documentation (above) if you would like to learn more.

3 Rendering objects as components

As we have seen up until now React allows us to create components which we render to the screen, based on the information (or state) within it. Whilst so far, we have simply used one component per page, following proper software practises such as encapsulation we can store all the information related to displaying any object type in a component of its own. We can then render this component whenever and wherever we like with any object of that type.

This is a very useful tool for working with sets/lists of data, as we only need to define the logic, layout, and styling once. You likely come across this practice daily in your internet browsing, one common example is YouTube (but almost every large application out there does this).



Here we can see that each of the videos has a thumbnail image, user profile picture, title, view count, and how long ago it was uploaded.

Now if we think back to the last lab, you can probably see that we could technically display these results in a table. However, you should intuitively know this probably wouldn't lead to the same level of user experience.

3.1 Exercise 3.1: Rendering objects with components using props

So far when we've inserted one of our components using the HTML tag we have done so without specifying any extra information. Within our tag we can define values called 'props' by name and pass in extra information.

For this example, we will replace the user table with a series of user components.

1. Create a new file ‘UserList.tsx’ this will be a stripped-down version of the ‘Users.tsx’ file. Within this file we will update how we loop through each user (similar to when we created the table), however this time we will be extracting that data out to our new component so we simply add a UserListObject component and set the user prop to the current user object.

```
import axios from 'axios';
import React from "react";
import CSS from 'csstype';
import {Paper, AlertTitle, Alert} from "@mui/material";
import UserListObject from "./UserListObject";

const UserList = () => {
   const [users, setUsers] = React.useState<Array<User>>([])
   const [errorFlag, setErrorFlag] = React.useState(false)
   const [errorMessage, setErrorMessage] = React.useState("")

   React.useEffect(() => {
       const getUsers = () =>  {
           axios.get('http://localhost:3000/api/users')
               .then((response) => {
                   setErrorFlag(false)
                   setErrorMessage("")
                   setUsers(response.data)
               }, (error) => {
                   setErrorFlag(true)
                   setErrorMessage(error.toString() + " defaulting
to old users changes app may not work as expected")
               })
       }
       getUsers()
   }, [setUsers])

   const user_rows = () => users.map((user: User) =>
<UserListObject key={user.user_id + user.username} user={user}/>)

   const card: CSS.Properties = {
       padding: "10px",
       margin: "20px",
       display: "block",
       width: "fit-content"
   }

   return (
       <Paper elevation={3} style={card}>
           <h1>UserList</h1>
           <div style={{display:"inline-block", maxWidth:"965px",
minWidth:"320"}}>
               {errorFlag?
                   <Alert severity="error">
                       <AlertTitle>Error</AlertTitle>
                       {errorMessage}
                   </Alert>
               :""}
               {user_rows()}
           </div>
       </Paper>
```

```
    )
}
export default UserList;
```

2. Now we need to define the UserListObject component, we will keep it simple for now. Here we take in a props object, and define a new state variable user based on the props.user value.

```
import React from "react";

interface IUserProps {
   user: User
}

const UserListObject = (props: IUserProps) => {
   const [user] = React.useState<User>(props.user)
   return (
       <h3>{user.username}</h3>
   )
}

export default UserListObject
```

3. Finally, we need to hook this up to our Router in App.tsx, adding this line above our /users/:id route

```
<Route path="/users-props" element={<UserList/>}/>
```

4. Run your application and go to “localhost:8080/user-props” and check that the users are being displayed as simple headings. **Note:** You may have to restart your application to see new routes if they were added whilst the application was running.

We will continue fleshing out this example in the next section, but importantly here we were able to use props to display a list of objects.

4 State management

In our examples so far, we have made use of React’s built in useState function. However, you may notice that we can only access this from within the current component (or children where we provide this as a prop). In many cases this functionality works perfectly fine as components should be broken up in a way that they only need access to their own (related) information. However, there are some instances where we want a global state, an example you will likely encounter within the scope of your second assignment is user details. For any secure site once a user logs in it needs to keep track of information that identifies and authorises the user (such as a token).

4.1 Zustand

There are many state management libraries available for use with React. The most popular is Redux however that library is verbose and complex, requiring a lot of boilerplate code. As a compromise the lab will show how we can store state with Zustand (https://www.npmjs.com/package/zustand), a newer and much more intuitive library for beginners.

Be warned that in this section we will use state management in a scenario that may be better achieved using other methods. For instance, we can pass functions as props so that a child element has access to select methods of its parent. When working on the assignment we suggest avoiding the use of global state for large data structures that do not have a wide scope. Meaning that a list of elements probably doesn't need to use global state if the list is only used by a parent and child component. However, if you were keeping track of the user who is currently logged in, this is a small amount of data that you will likely need all throughout your application, so a global state makes sense.

4.1.1 Exercise 4.1: Creating a state store

To start install Zustand with npm using the following command *'npm install --save zustand'* Within the domain of state management, the term 'store' refers to a place where we keep and manage our state. Create a folder "store" within your "src" directory. Within this folder create a file "index.ts" and copy the following code.

```
import create from 'zustand';

interface UserState {
   users: User[];
   setUsers: (users: Array<User>) => void;
   editUser: (user: User, newUsername: string) => void;
   removeUser: (user: User) => void;
}

const useStore = create<UserState>((set) => ({
   users: [],
   setUsers: (users: Array<User>) => set(() => {
       return {users: users}
   }),
   editUser: (user: User, newUsername) => set((state) => {
       return {users: state.users.map(u => u.user_id === user.user_id ?
({...u, username: newUsername} as User): u)}
   }),
   removeUser: (user: User) => set((state) => {
       return {users: state.users.filter(u => u.user_id !==
user.user_id)}
   })
}))

export const useUserStore = useStore;
```

4.1.2 Exercise 4.2: Using our state

Now that we have created our state we can easily access the value itself and the functions we defined to change it from other components. We will adapt and continue our previous example with the use of state. In UserList.tsx replace your users state definition with

```
import {useUserStore} from "../store";
...
const users = useUserStore(state => state.users)
const setUsers = useUserStore(state => state.setUsers)
```

Within our UserListObject we can then access the state similarly to as we have done above, this time using the edit and remove functions. Below is the code for the UserListObject component making use of the MUI Card component. Cards are quite a standard way to show information in material design, and make use of 3 different sections: media, content, and actions. You will have to add in your own edit and delete dialogs where stated, these may require some minor changes from what we have done previously.

```
import React from "react";
import axios from "axios";
import {Delete, Edit} from "@mui/icons-material";
import {useUserStore} from "../store";
import {Button, Card, CardActions, CardContent, CardMedia, Dialog,
DialogActions, DialogContent, DialogContentText,
    DialogTitle, IconButton, TextField, Typography} from "@mui/material";
import CSS from 'csstype';

interface IUserProps {
    user: User
}

const UserListObject = (props: IUserProps) => {
    const [user] = React.useState<User>(props.user)
    const [username, setUsername] = React.useState("")
    const [openEditDialog, setOpenEditDialog] = React.useState(false)
    const [openDeleteDialog, setOpenDeleteDialog] = React.useState(false)
    const deleteUserFromStore = useUserStore(state => state.removeUser)
    const editUserFromStore = useUserStore(state => state.editUser)

    const deleteUser = () => {
        axios.delete('http://localhost:3000/api/users/' + user.user_id)
            .then(() => {
                deleteUserFromStore(user)
            })
    }
    const editUser = () => {
        axios.put('http://localhost:3000/api/users/'+user.user_id,
{"username": username})
            .then(() => {
                editUserFromStore(user, username)
               })
    }

    const userCardStyles: CSS.Properties = {
        display: "inline-block",
        height: "328px",
        width: "300px",
        margin: "10px",
        padding: "0px"
    }

    return (
        <Card sx={userCardStyles}>
            <CardMedia
                component="img"
                height="200"
                width="200"
                sx={{objectFit:"cover"}}
                image="https://atasouthport.com/wp-
```

```
content/uploads/2017/04/default-image.jpg"
                alt="User hero image"
            />
            <CardContent>
                <Typography variant="h4">
                    {user.user_id} {user.username}
                </Typography>
            </CardContent>
            <CardActions>
                <IconButton onClick={() => {setOpenEditDialog(true)}}>
                    <Edit/>
                </IconButton>
                <IconButton onClick={() => {setOpenDeleteDialog(true)}}>
                    <Delete/>
                </IconButton>
            </CardActions>
ADD EDIT/DELETE DIALOGS HERE
        </Card>
    )
}
export default UserListObject
```

Now as we interact (edit or delete) with each card we see that the whole list updates even though we are not explicitly changing this list in the UserList UserListObject components, instead we change the state which makes UserList re-render the components that have changed. **Note:** In this case it can tell which ones have changed based on the key we provide (this is why we include the username, otherwise it would not re-render the UserListObject, since it would believe it was the same and not want to waste resources reloading the component)

4.1.3 Exercise 4.3: Persisting our state

With the state we have created so far, you likely noticed that we lose our state as soon as we refresh the page. For some use cases this is fine, though for others we might want this state to persist. If that is the case then we can make use of the browsers local storage to save our state every time it gets changed, and read it in when the application initialises the state.

In our store/index.ts file we can replace the code with the updated code below. **Note:** We define functions here to interact with the local storage, these handle converting our list of users to and from a string, as the underlying local storage mechanism only allows for storing key value pairs of strings (https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage).

```
import create from 'zustand';

interface UserState {
    users: User[];
    setUsers: (users: Array<User>) => void;
    editUser: (user: User, newUsername: string) => void;
    removeUser: (user: User) => void;
}
const getLocalStorage = (key: string): Array<User> =>
```

```
JSON.parse(window.localStorage.getItem(key) as string);
const setLocalStorage = (key: string, value:Array<User>) =>
window.localStorage.setItem(key, JSON.stringify(value));

const useStore = create<UserState>((set) => ({
   users: getLocalStorage('users') || [],
   setUsers: (users: Array<User>) => set(() => {
       setLocalStorage('users', users)
       return {users: users}
   }),
   editUser: (user: User, newUsername) => set((state) => {
       const temp = state.users.map(u => u.user_id === user.user_id ?
({...u, username: newUsername} as User): u)
       setLocalStorage('users', temp)
       return {users: temp}
   }),
   removeUser: (user: User) => set((state) => {
       setLocalStorage('users', state.users.filter(u => u.user_id !==
user.user_id))
       return {users: state.users.filter(u => u.user_id !==
user.user_id)}
   })
}))

export const useUserStore = useStore;
```

Now if you run the application once with the server on, then turn your server off you should see that the user list is still displayed. While in this example this functionality is not very useful as we cannot manipulate the data, more advanced web pages can keep track of changes made ‘offline’ and then commit these later. One common example of this is Google docs.

5 Final thoughts

After having made your way through this lab you have hopefully gained more appreciation for creating better looking, reusable, and versatile React components. From here you should be ready to tackle the second assignment. One useful trick is to draw out a draft interface, and decide how each different component fits into this (make sure to think about the flow of data within and between these components). Finally, make sure to make use of online documentation and examples when making use of libraries. This is particularly important with UI component libraries, as a good look at the documentation and examples may give you good ideas of how to display something, and often even within one library there are multiple ways to achieve similar results. So, having a deeper understanding can reduce the chance you pick a sub-optimal option.