# Table of Contents


PART 01 – FOCUSSING WITH THE USEREF HOOK ..... 2

PART 02 – POST REGISTRATION ..... 3

PART 03 – LOGGING OUT ..... 4

PART 04 – ADDING NEW WEIGHTS ..... 9

PART 05 – HANDLIING WEIGHT DATA ..... 12

PART 06 – THE USECALLBACK AND USEMEMO HOOKS ..... 15

PART 07 – (OPTIONAL) CODE IMPROVEMENTS ..... 18

APPENDIX A – THE USEREF() HOOK ..... 21

## PART 01 – FOCUSSING WITH THE USEREF HOOK

We will now introduce another hook that is used mainly with the DOM. This hook does NOT force a re-render if it's value changes, so no performance hit. That means it can be modified and updated without a re-render. There is a special property of this hook called `current`, this will give us access to the latest value of whatever it is we are storing.

Two good use cases for this hook is to focus on a form field and scroll back to the top of the view (page). Use the `login` component but it applies to the `register` component or any form.

Of course, we could just use the browser native `autoFocus` attribute, so this example is for demonstration purposes only.

1. Start by de-strucuting the necessary functions so `useRef()`. However, in this use case context, we do need another trigger to initiate the `useRef` functionality. When the component is first rendered, that's when the `useRef()` object is initialized. We need a trigger for it to perform its task so we use `useEffect()` for this situation. The task is to focus to a field on the form:

```
import { useState, useContext, useRef, useEffect } from "react";
import { useNavigate } from "react-router-dom";
import { AuthContext } from "../AuthContext";
```

2. Initialize the useRef() hook:

```
  const { login } = useContext(AuthContext);
  const focusUser = useRef(null);
  const handleFieldChange = (event) => {
```

Initially the `focusUser` object is `null`, as a starting reference. We are not using the value of the field, just the field itself, as a referene to the form control.

3. Now we will assign a reference to the `username` DOM element which gives us access into the DOM node itself:

```
          <input
            type="text"
            name="username"
            value={username}
            onChange={handleFieldChange}
            ref={focusUser}
          />
        </label>
      </div>
```

So now, `focusUser` refers to the `username` form field.

4. Finally to ensure that this reference is made at least once (on form load), we add the logic we need, we encorporate the `useEffect()` hook:

```
useEffect(() => {
  focusUser.current.focus();
}, []);
```

This function can go above the `handleFieldChange()` function literal. Remember that the empty array after the innerfunction of `useEffect()` means that the function runs only once.

To check that the functionality works, first go to `/login` and you will see the cursor appear immediately in the Name field. Now nagivate to `/register` and you will not see the cursor in the Name field.

## PART 02 – POST REGISTRATION

In this part, we will login the newly registered user. We will assume that if you register, you are automatically logged in. In this case, we will not use login status, there is no need as yet. We do need to re-direct the user to the home view however. We will then create a log out process.

Note, this is the automatic login after registering on the site. The entire login process was discussed in Day01 of the bootcamp.

Before adding these new features we need some clean up. We need to prevent the re-submit of values, disable the submit button and clear the values from both fields. Lets start with re-directing the user to the `home` view, so import `useNaviate`.

1. We will also import all the packages we need so, `useContext`, and `AuthContext` in the register component:

```
import { useState, useContext } from "react";
import { useNavigate } from "react-router-dom";
import { AuthContext } from "../AuthContext";
```

We are doing this in the `register` component the main.jsx file.

2. Create variables for holding the state of form submit. Just like with the login component, we need to de-sctucture the `login` function from `AuthContext`. Finally for this section we create a `navigate` object:

```
function Main( ) {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");
  const [isSubmitting, setIsSubmitting] = useState(false);
  const { login } = useContext(AuthContext);
  const navigate = useNavigate();
```

3. We have the `login()` function from `AuthContext`, so just call it inside of `handleSubmit()` and pass the current `username` and `password`, as an object. We are in submitting (switch to true) and send user to `home`:

```
      })
        .then(response => response.text())
        .then(emp => {
          login({ username, password });
          setUsername("");
          setPassword("");
          setIsSubmitting(true);
          console.log(emp);
          navigate("/");
        })
        .catch(err => {
```

Its not necessary to log the user, so if you delete that line, then no need for the parameter `emp` either.

4. Adjust the buton to reflect the status of submitting or not submitting:

```
          </label>
        </div>
        <button type="submit" disabled={isSubmitting}>
          Submit
        </button>
```

Now if you register a new user, they would automatically be shown as logged in. Of course this is a beginner bootcamp, so I am not performing security checks and other advanced security screening.

## PART 03 – LOGGING OUT

The login process may not be industry standard but it works and we are learning from it. The architecture is weird but lets go with it. We will create a logout file (component) to handle logging out. The user will click on her name in the menu and a pop up will appear asking for permission to logout.

1. Create a new file in the wrapper folder called popup.jsx. When the function is defined, there are no parameters but this will change shortly:

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

    </div>
  );
};
export default Popup;
```

Another component will call this function soon. Notice the exported default.

2. This is the structure of the popup file. Only the last pair of `div` tags and the `p` tags are important, however due to the complexity of the CSS, we need two levels of parent `divs`:

```
return (
  <div >
    <div >
      <p></p>
      <div>
        <button >Cancel</button>
        <button>Logout</button>
      </div>
    </div>
  </div>
);
```

The `<p>` tag will hold the message, see #4

3. Now just add the necessary CSS classes:

```
return (
  <div className="popup-overlay">
    <div className="popup-content">
      <p> </p>
      <div className="popup-actions">
        <button >Cancel</button>
        <button className="popup-logout" >Logout</button>
      </div>
    </div>
  </div>
);
```

The styles.css file in the downloaded files, will contain all the classes used here. The styles.css file is in the src folder for Part 03.

4. Add a `message` property between the `p-tags`, the value of this message will be passed into this popup component from the calling component:

```
<div className="popup-content">
  <p>{message}</p>
  <div className="popup-actions">
```

Note, it is the *value* of message that will display between the <p> tags.

The popup.jsx file is triggered from the `Header` component. In the header component we use a state variable called `showPopup` to decide if and when to show the popup.jsx component. Once the logged in name is clicked, an event handler will set `showPopup` to true. We will also define the `message` to pass to popup, **the first of three parameters** (#5). The other two parameters passed to popup are actually references to functions in the header component! These functions are activated from the `popup` component. Yes, you read that right.

In this approach, we can reuse the `popup` component. It relies entirely on it's caller component for its behavior. In this case, the `header` component calls and controls the `popup` component. In this case we use it for logging out.

5. Back to the parameter list mentioned above. Remember this popup.jsx file can be called from *any* component. It is built to be reusable. Therefore we need a way to control its behavior, the behavior of the popup. It makes sense therefore to configure this popup via it's parameters. This is the only way we have to pass data and objects into the popup.jsx file:

```
function Popup( { message, onConfirm, onCancel } ) {
  return (
    <div>
```

Now we have a way to pass a message, any message into this popup component. The CSS will take care of the display. The other two parameters is to handle the buttons on this popup, the user will click OK or CANCEL or something similar. We have to handle these events and the code to handle the events will be in the calling component, but triggered from the child component, popup.jsx in this case. Notice that the parameter is a *single* object with *three* properties.

6. Now that we have access to the parameter object, we can now call funcitons on the parent! Lets wire up `onConfirm` and `onCancel` in the popup child:

```
<p> {message} </p>
<div className="popup-actions">
  <button onClick={onCancel}>Cancel</button>
  <button className="popup-logout" onClick={onConfirm}>Logout</button>
</div>
```

Remember, these functions are on the parent.

7. The popup component is just like any other component, it is exported so we can import it into the `Header` component:

```
import { useContext, useState } from "react";
import { AuthContext } from "../AuthContext";
import Popup from "./popup";
//
function Header() {
```

Note, since there is only one function which is the default function in the popup file, when we import, no need to use curly braces. Also we must keep track of the popup component, so we import and implement the `useState` hook.

8. Add state for the popup itself, so a Boolean value will control wheather the popup component is shown or not:

```
function Header() {
  const { user, logout } = useContext(AuthContext);
  const [showPopup, setShowPopup] = useState(false);
```

We also need to destructure the `logout()` function from `AuthContext`. Remember we established the authorization context in Day01, part 06.

9. From #7 the `showPopup` state is initially set to `false`, but we want to turn it on when the user clicks on her name, so lets add a function for that:

```
function Header() {
  const { user, logout } = useContext(AuthContext);
  const [showPopup, setShowPopup] = useState(false);
  const handleLogoutClick = () => setShowPopup(true);
```

This will activate the popup component when the user clicks her name in the header part of the view. The popup component will show below the menu. Of course we need to configure this show/not show from the JSX code, see #10.

10. Since we have this state property called `showPopup`, we can use it to either show or not show the `Popup` component on the browser. This component relies heavily on CSS styles to position it correctly:

```
      </nav>
      { showPopup && (<Popup />) }
    </header>
```

I am using short-circuiting here, it's a JS concept.

11. Now, remember those three parameters in the object that is passed to the actual popup.jsx file, well here is where we can add our custom message along with the behavior of the buttons on that popup.jsx file:

```
      </nav>
      {showPopup && (
        <Popup
          message="Are you sure you want to logout?"
          onConfirm={handleConfirmLogout}
          onCancel={handleCancelLogout}
        />
      )}
    </header>
```

So, in this way, our message can be any string. It is customizable. Notice also that we handle the OK or CANCEL from the parent component, NOT the popup itself. The popup component only activates either of these buttons via the user.

12. As usual we should confirm that the user wants to logout. If she chooses to logout we should call the `logout()` function in the `AuthContext` context and change the status of the `showPopup` to false:

```
  const handleLogoutClick = () => setShowPopup(true);
  const handleConfirmLogout = () => {
    logout();
    setShowPopup(false);
  };
```

Remember from `authcontext`, when we call the `logout()` function, we are actually setting the user to `null`. The function `setShowPopup()` is handled by the `useState()` hook in the header.jsx file.

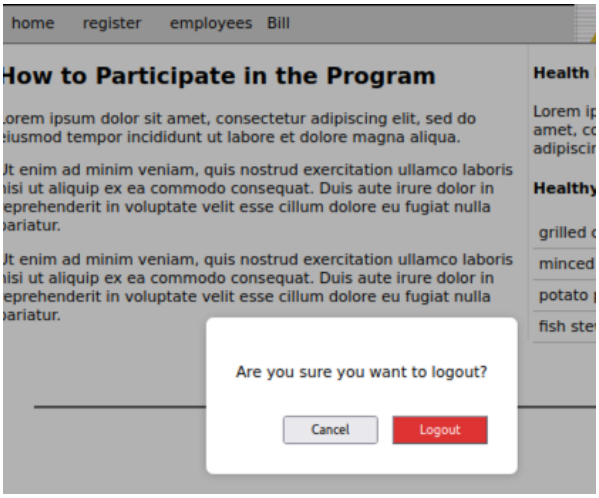
13. Lets not forget that the `popup` component has a cancel button on it. We also have a function associated with that button called `handleCancelLogout()`. We need to define that function now:

```
    setShowPopup(false);
  };
  const handleCancelLogout = () => setShowPopup(false);
  return (
    <header>
```

So, all this function does is to remove the popup component from the browser. Remember the actual disappearing act is handled by the JSX code on the HTML side. The property `showPopup` can be toggled.

14. Finally add the event to call the `handleLogoutClick()` function located at the top of the header.jsx file and to get the component un-mounted from the browser:

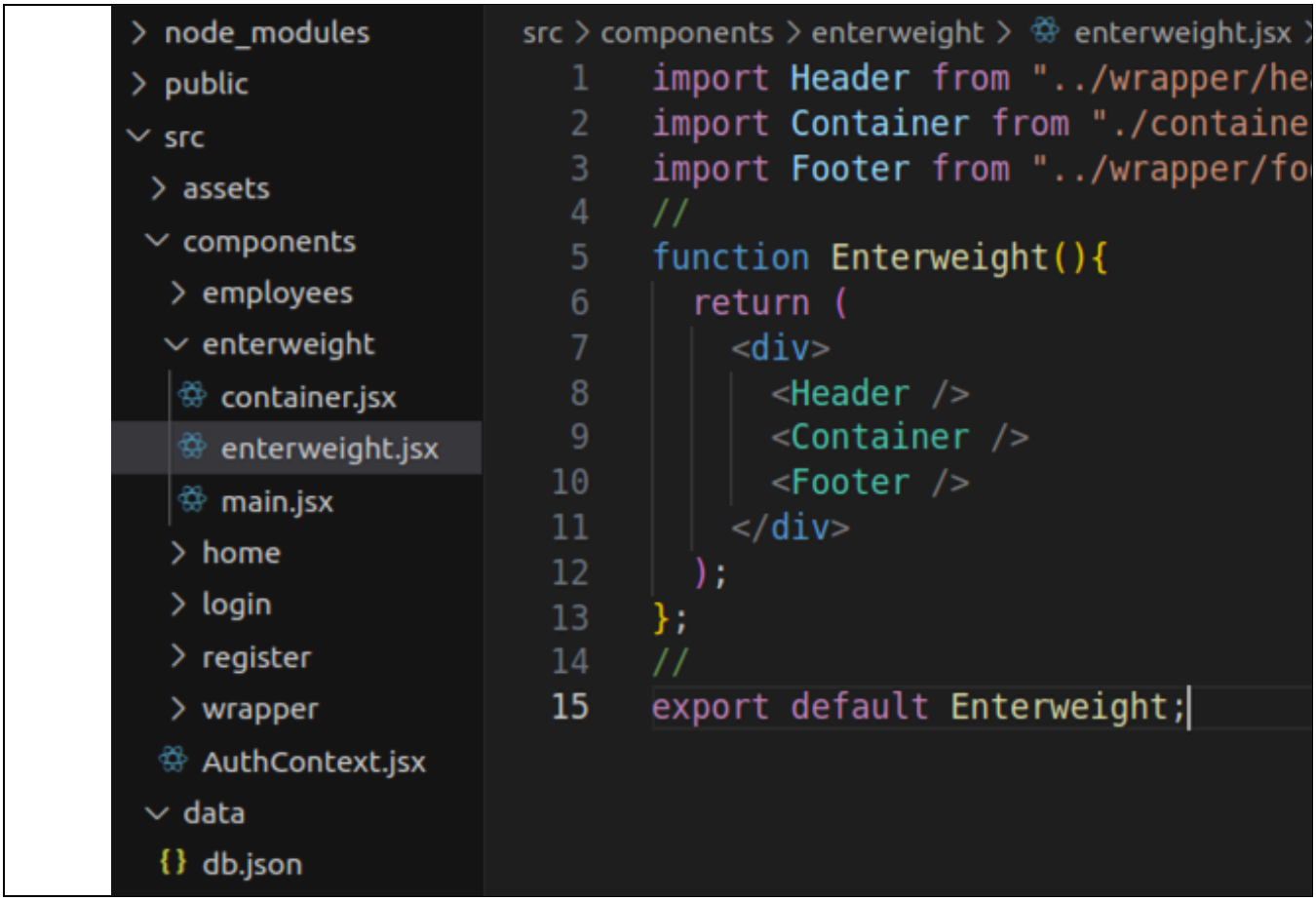
```
          <li><NavLink to="/employees">employees</NavLink></li>
          {user ? (
            <li onClick={handleLogoutClick}>{user.username}</li>
          ) : (
            <li><NavLink to="/login">login</NavLink></li>
          )}
        </ul>
      </nav>
```



## PART 04 – ADDING NEW WEIGHTS

In this section we will allow the user logged in to add her new weight. The weight numeric value will be the only thing we need. We already have the date and the user who is logged in, so with the weight we have all three pieces of data we can document for that individual. This gives us an opportunity to dive deeper into validation.

1. Start by copying the register component since it already has lots of details we need for the enter weight component




2. In the main.jsx file in the enterweight folder, change the heading, we will insert the actual date later:

```
return (
  <main>
    <h2>Enter your weight for today</h2>
    <form onSubmit={handleSubmit}>
      <div>
```

3. The App.jsx has to be updated to route to this new component:

```
import Login from "./components/login/login";
import Enterweight from "./components/enterweight/enterweight";
...
function App() {
  return (
...
        <Route path='/login' element={<Login />} />
        <Route path='/enterweight' element={<Enterweight />} />
      </Routes>
    </BrowserRouter>
```

4. Of course we add a new menu item in the `header` component, it can be placed to the left of the login link:

```
<li><NavLink to="/register">register</NavLink></li>
<li><NavLink to="/employees">employees</NavLink></li>
<li><NavLink to="/enterweight">enter weight</NavLink></li>
{user ? (
  <li onClick={handleLogoutClick}>{user.username}</li>
) : (
```

At this point, spin the app and make sure you can navigate to the new component

5. On the form itself, we can begin by removing everthing that has to do with the `password` field, we will then rename the `username` field to just `userweight`. In a case like this, it is better to start where the last place that password appears and work backwards to the top of the main.jsx file in the enterweight folder.

6. Also remove anything that refers to logging in (except `AuthContext`), we are already logged in at this point. We will remove all login/logout code. Again work backwards.

7. If user is not logged in, lets show an appropriate message, if they got to this view without logging in. First we get the user property from `AuthContext`:

```
const [isSubmitting, setIsSubmitting] = useState(false);
const { user } = useContext(AuthContext);
const navigate = useNavigate();
```

8. Now we use the `user` property to either show or not show the form:

```
return (
    <main>
      {user ? (
        <>
          <h2>Enter your weight for today</h2>
          <form onSubmit={handleSubmit}>
            <div>
              <label>
                Today's weight:
                <input
                  type="text"
                  name="userweight"
                  value={userweight}
                  onChange={handleFieldChange}
                />
              </label>
            </div>
            <button type="submit" disabled={isSubmitting}>
              Submit
            </button>
          </form>
        </>
      ) : (
        <h4>You must be logged in to post a weight</h4>
      )}
    </main>
        );
```

9. We will need to change the `username` field to `userweight`, so change all occurances of this field name:

```
function Main( ) {
  const [userweight, setUserWeight] = useState("");
  const [isSubmitting, setIsSubmitting] = useState(false);
  const { user, login } = useContext(AuthContext);
```

   This is just the first occurance of username being changed to weight, there are several more instances.

Since there were a lot of changes here, it is better to start the app and make sure all links still work before moving on. Do not add a weight as yet, we will do that shortly.

## PART 05 – HANDLIING WEIGHT DATA

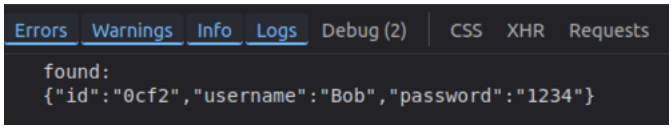
We would need a new array in the .json file for weights. I have added a *weights* section and it already contains at least five different entries for each of the three original employees Axle, Bob and Jane. Basically we need the *user*, *date* of entry and the *weight* itself, in Kgs. One Kg is approximately 2.2 lbs. Note, this was done so that the json-server can work with just the **one** file but **two** endpoints.

This new db.json file is included in the downloaded files for today.

1. Initially when we logged in a user, we never stored the id, even though we had access to it. Lets make sure we have all the pieces before moving on. In the Login component, in main, add this line:

```
        user => user.username === username && user.password === password);
        if (found) {
          login(found);
          console.log("found: "+JSON.stringify(found));
          setLoginStatus("Login successful!");
```

You should now get an output like this one below:




This proves that `found` contains all the data we need to proceed. You can now remove the highlighted line.

2. Now back to the main.jsx file in the enterweight component. Remember we copied the `login` component, so the endpoint is now incorrect for **this** file, lets change it to `/weights` to match our new db.json file section:

```
    fetch('http://localhost:3030/weights', {
            method: 'POST',
            headers: {
              'Accept': 'application/json',
              'Content-Type': 'application/json'
            },
```

4. We need the date in the format dd/mm/yyyy. Here we use regular JS code to get the date and convert to format. Now enter this code just above the `fetch` but below the `handleSubmit()`:

```
      const handleSubmit = (event) => {
          event.preventDefault();
          const today = new Date();
          const day = String(today.getDate()).padStart(2, '0');
          const month = String(today.getMonth() + 1).padStart(2, '0');
          const year = today.getFullYear();
          const formattedDate = `${day}/${month}/${year}`;
```

5. The weights section of our json file requires three key/value pairs:

```
        },
        body: JSON.stringify({
          id: user.id,
          weight: userweight,
          date: formattedDate
        })
      })
        .then(response => response.json())
```

```
"weights": [
  {
    "id": "a4fb",
    "weight": "70",
    "date": "18/11/2024"
  },
```

You may already have the userweight key/value pair. Just verify the spelling.

Lets recap the above, the `user.id` comes from the authorization context. The `userweight` comes from what the user just entered. The `formattedDate` comes from the JS date conversion we did at #4. This can be any date format you choose.

6. We might want to consider adding some validation to the value being entered in by the user. Lets make sure the value is **not null, is numeric and is between 20 and 250 kg**. First lets setup a state variable to hold any errors:

```
function Main( ) {
  const [weight, setweight] = useState("");
  const [isSubmitting, setIsSubmitting] = useState(false);
  const [error, setError] = useState("");
  const { user } = useContext(AuthContext);
  const navigate = useNavigate();
```

7. Most of the validation code will appear just after the `handleSubmit()` definition:

```
  const handleSubmit = (event) => {
    event.preventDefault();
    setError("");
    if (weight === "") {
      setError("Weight is required.");
      return;
    }
    const numWeight = Number(weight);
    if (isNaN(numWeight)) {
      setError("Weight must be a number.");
      return;
    }
    if (numWeight < 20 || numWeight > 250) {
      setError("Weight must be between 20 and 250.");
      return;
    }
```

Remember that all state variables are essentially updated, they are not just assigned.

8. We can display the actual error message (if any) just above the submit button. I included the CSS style here as it is very simple:

```
                    </div>
                    {error && <p style={{ color: 'red' }}>{error}</p>}
                    <button type="submit" disabled={isSubmitting}>
                      Submit
                    </button>
```

The following few points are optional (#9 to #11). This is just additional checks we can add just after the `fetch()` method to make our application a bit more robust.

9. Before anything we can create a payload of all the key/value pairs we need to pass over to the mock database. Add this code just before the `fetch()`:

```
        const payload = {
          id: user.id,
          weight: String(weight),
          date: formattedDate
        };
```

Note, React does not currently have the concept of an interface. A plain JS `enum` can be used. React does use the `PropTypes.shape()` library function to define an object's structure.

Change the `body` part of the `fetch()` method:

```
        fetch('http://localhost:3030/weights', {
          method: 'POST',
          headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
          },
          body: JSON.stringify(payload)
        })
```

As a separate `payload`, we can add more security in the future.

10. When the `fetch()` method executes, it engages with an external server. We may want to add code to ensure that the transaction went as planned, if not, lets deal with the consequences like server not found:

```
          body: JSON.stringify(payload)
        })
        .then(response => {
          if (!response.ok) {
            if (response.status === 404) {
              throw new Error("Resource not found. The server endpoint may be
incorrect.");
            }
            throw new Error(`Server error: ${response.status}`);
          }
          return response.text();
        })
        .then(() => {
```

Once the handshake happens we can continu and return the text part of our POST.

11. Next we can attempt to parse the response, just in case it is empty or some other error occurred:

```
            return response.text();
          })
          .then(text => {
            if (text) {
              try {
                JSON.parse(text);
              } catch (e) {
                throw new Error('Invalid JSON response: ' + text);
              }
            }
          })
          .then(() => {
```

To test this new *enter weight* feature, first log in as one of the users in the `employees` database. After logging in, you will be taken to the `home` view. Navigate to the `enter weight` view and enter a weight for whoever is logged in. Then go into the db.json file and the weight you just entered and the user's id will be at the bottom of the db.json file.

## PART 06 – THE USECALLBACK AND USEMEMO HOOKS

In this type of development, there is a lot of re-rendering going on. A re-render is when React must re-paint the section of the file related to the `render()` function. For example in the header.jsx file (in the wrapper folder) the <header> component has to re-render when the user logs in or out.

Well, react has come up with a way to cache entire functions so that they do not have to be re-recreated each time the `<header>` component re-paints itself on the browser screen. This saves on memory usage.

The three functons in this header file, so `handleLogoutClick()`, `handleConfirmLogout()`, and `handleCancelLogout()` functions can be wrapped within `useCallback()`.

This is to avoid re-creating them on every render. Also we pass two of them to the Popup component. `useCallback()` here prevents the popup component from re-rendering unnecessarily.

Lets add this feature to the `Header` component.

1. First we need to import the hook, so we get that from the react package:

```
import { NavLink } from "react-router-dom";
import {useContext, useState, useCallback} from "react";
import {AuthContext} from "./../AuthContext";
```

2. Here is the first use of `useCallback()` with the `handleLogoutClick()` function:

```
  const handleLogoutClick = useCallback(() => {
    setShowPopup(true);
  }, []);
```

   This means that this function will not be re-created each time this component is rendered on the browser. The function `setShowPopup`() will operate normally.

3. In the case of `handleConfirmLogout`(), it does have a dependency, this function depends on the state of what happens in `logout()`:

```
  const handleConfirmLogout = useCallback(() => {
    logout();
    setShowPopup(false);
  }, [logout]);
```

   If the value of `logout` changes, the `handleConfirmLogout()` function *is* re-created in memory. Notice that `logout` is placed inside of the array brackets at the end. The function always uses the latest value of its dependencies. Remember that `logout()` in `AuthContext` changes the `user` object to null.

2. Here is the final function being cached:

```
  const handleCancelLogout = useCallback(() => {
    setShowPopup(false);
  }, []);
```

   An empty array means no dependencies

3. If you remember, we used the `fetch()` method to return all the employee data in the db.json file. Well what if we had to sort that data or filter the data. Also if we had thousands of documents, these would be very expensive actions. Here we can use the `useMemo` feature from React to cache the returned data and perform these sorting and filtering actions in memory. To use `useMemo`, first de-structure the function from react:

```
import { useEffect, useState, useMemo } from "react";
function Main() {
```

   Do this in the main.jsx file in the employees folder.

4. Now, just before the `return` function create a property that points to the `useMemo` function, then populate the `allEmployees` data inside of `useMemo`:

```
}, []);
const sortedEmployees = useMemo(() => allEmployees, [allEmployees]);
return (
```

Note, there are two parameters in the `useMemo` function, another function and an array. That first parameter, the function, is the function that performs the expensive calculation like getting large amounts of data.

The second parameter, the array is used by the first parameter. So, the first parameter, the function, depends on this second parameter, the array of values.

In our case, this second parameter, the array would have been loaded when the component first rendered. It is NOT recommended to use useMemo() just after the `fetch()` executes. The `useMemo()` hook is typically used with state objects. In this case allEmployees is a state object.

We could just return the `allEmployees` array here, but it wont give us any benefit, usually the array is sorted, or worked on in some way before being returned. This is where the benefit happens, see below.

5. If instead we wanted to sort the employees, we could now use this construction:

```
const sortedEmployees = useMemo(() => {
  return [...allEmployees].sort(
    (a, b) => a.username.localeCompare(b.username)
  );
}, [allEmployees]);
```

Here we make a deep copy of everything in `allEmployees`, then call the `sort()` function on that data. Finally the sorted data is returned. This sorting will only happen again if **and only if** an array element changes.

The main purpose of `useMemo` is performance optimization. We avoid expensive or time-consuming calculations every time the component renders. If the employee list in this case did not change, then there is no need to execute the sorting operation.

6. Then in the browser just show the sorted array:

```
sortedEmployees.map((customer, i) =>
  (<div key={i}>
    {customer.username} 
```

## PART 07 – (OPTIONAL) CODE IMPROVEMENTS

When I created this bootcamp, learning was key to the decisions I made. This means that I did not really pay attention to details as those attention to details would hav distracted the learner from focusing on the key concepts. Once the key concepts have been understood, then all the other details like security, error handling, performance and accessibility can be added.

I am going to use the last file we worked on, the main.jsx in the employeed component to add some of these details. Lets start with better error handling.

1. First we need to add state management for errors, as there can be several:

```
function Main() {
  const [allEmployees, setAllEmployees] = useState([]);
  const [error, setError] = useState(null);
```

2. Make sure that just before the `fetch()` the erorrs object is null:

```
    useEffect(()=>{
      setError(null);
      fetch("http://localhost:3030/employees")
```

3. Once the data has been resolved ( or not) add the specific error message to the errors object. Of course you can also log the message :

```
  const handleConfirmLogout = useCallback(() => {
      .then(resolvedData => {
        setAllEmployees(resolvedData);
      })
      .catch((error)=>{
        console.error("Error fetching employees:", error.message);
        setError(`Failed to load employees: ${error.message}`);
      });
```

   Notice the use of JS template literals here.

4. Just before the `return()` check for errors and display errors instead of the normal content:

```
    if (error) {
      return (
        <main>
          <h2>Employee List</h2>
          <div className="error">{error}</div>
          <button onClick={() => window.location.reload()}>Try Again</button>
        </main>
      );
    }
    return (
      <main className="employees-main">
```

   The `error` object will be cleared when the component re-renders.

5. We can add a loading message in the same way. With this feature, if there is a lot of data to load and its taking a long time, the user will see a loading message:

```
function Main() {
  const [allEmployees, setAllEmployees] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
```

6. Just like for the error object, toggle the loading object to true once inside the useEffect(), we are about to get the data:

```
  useEffect(()=>{
    setLoading(true);
    setError(null);
    fetch("http://localhost:3030/employees")
```

7. Once the data has been resolved toggle the loading object to false, we are done for now :

```
      return data.json();
    })
    .then(resolvedData => {
      setAllEmployees(resolvedData);
      setLoading(false);
    })
```

8. If the data was not resolved we should also toggle the loading object to false :

```
      setLoading(false);
    })
    .catch((error)=>{
      console.error("Error fetching employees:", error.message);
      setError(`Failed to load employees: ${error.message}`);
      setLoading(false);
```

9. Here is the actual display for the loading message on the view:

```
  }, [allEmployees, searchTerm]);
  if (loading) {
    return (
      <main>
        <h2>Employee List</h2>
        <div className="loading">Loading employees...</div>
      </main>
    );
  }
  if (error) {
    return
```

10. We can improve the display as well because there is a chance we do not get any dat from our fetch():

```
    {sortedEmployees.length === 0 ? (
      <div className="no-data">No employees found.</div>
    ) : (
      <div className="employees-grid">
        {sortedEmployees.map((employee) =>
          (<div key={employee.id || employee.username} className="employee-card">
            <h3>{employee.username}</h3>
            <span className="employee-status">Active</span>
          </div>)
        )}
      </div>
    )}
```

Before showing any data first check that the array is not empty. If empty the display is a simple div with an appropriate message.

If there is data we can put that data into some sort of table or grid. When iterating over the data we use the username as backup, just in case there is no id for the employee. We can add a CSS class for better presentation in the future.

Finally toward the bottom of each employee, add the Active flag there to show that the employee is still on payroll.

# APPENDIX A – THE USEREF() HOOK

The useRef() hook actually points to an object { current: initialValue } where you can store values you need to reference later but don't want to affect the component's render cycle.

When this hook is used, React creates an object in memory which keeps the initial value while the component is in focus. Also the value in this object does not cause a re-render if it changes.

This hook is mainly used for DOM elements. It is for working with the element itself, not its dynamic content. So, you cannot get the value that the use typed into an input box for example useState object.

Another use of useRef is to keep a value that needs to be accessed as the component re-renders, perhaps do to some external action. The object in useRef will keep its value, so timers are used in this manner. As the time changes, the value of the object changes but the component is not re-rendered.

If you need to compare a value from a previous state to the current one, useRef can be used for this purpose. Also you can create instance variables with this hook. Think of *static* variables in a class.