Table of Contents

[**part 01 – Adding New Weights** 2](#_Toc212463234)

[**part 02 – Handliing Weight Data** 6](#_Toc212463235)

[**part 03 – The useCallBAck and useMemo Hooks** 11](#_Toc212463236)

[**part 04 – Modernizing with Async/Await** 14](#_Toc212463237)

[**part 05 – Preparing for Child Routes** 18](#_Toc212463238)

[**part 06 – Child Routes** 19](#_Toc212463239)

[**part 07 – Admin Login Logic** 23](#_Toc212463240)

# part 01 – 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

|  |
| --- |
| **A screen shot of a computer screen  AI-generated content may be incorrect.** |

1. 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>** |

1. 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>** |

1. 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

1. 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 at the last place that *password* appears and work backwards to the top of the main.jsx file in the enterweight folder.
2. We are already logged in at this point, so remove any code that logs the user in (if any). We will need to verify that the user IS logged in as well as keep state on a few properties, so lets import the following:

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

1. Lets setup a few function variables to begin:

|  |
| --- |
| **function Main() {  const [userweight, setUserWeight] = useState("");**  **const [isSubmitting, setIsSubmitting] = useState(false);  const navigate = useNavigate();** |

We need to keep track of the numerical data for weight, that the user will insert. We also need to keep track of the submitting state, are we submitting or not. Of course we could ignore this but it is better to have it in there. Once the weight is submitted, we should re-direct them somewhere else, so that they don’t enter another weight.

1. 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 [userweight, setUserWeight] = useState("");**  **const [isSubmitting, setIsSubmitting] = useState(false);**  **const navigate = useNavigate();**  **const { user } = useContext(AuthContext);**  **const handleFieldChange = (event) => {** |

Once we have the user object, we have most of the details about the user.

1. Now we use the user property ( so user is not null) we can 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>**  **);** |

Note, we had username here since we copied the register component. Just replace username with userweight, we already know who is logged in at this point.

1. Check the object being posted, it must be weight and not username:

|  |
| --- |
| **},**  **body: JSON.stringify({**  **userweight: userweight**  **})**  **})**  **.then(response => response.text())** |

1. Lets turn to the isSubmitting Boolean value. It starts out as false when we declared it but we need to turn it to true if the button is clicked. We will need to remove the code about emp or the user and replace it with code about submitting:

|  |
| --- |
| **.then(response => response.text())**  **.then(() => {**  **setIsSubmitting(true);**  **navigate("/");**  **})** |

At this point we can re-direct them to the home view or some where else.

Note, the enter weights part of the app wont work properly at this point, we need to complete Part 02 for the actual posting of the weight.

# 

# part 02 – 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. 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!");** |

A screenshot of a computer

AI-generated content may be incorrect.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.

1. 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())** | **A screen shot of a computer code  AI-generated content may be incorrect.** |

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. Lets deal with the value the user is typing into the weight field. You already have a function called handleFieldChange(), lets adapt it for userweight:

|  |
| --- |
| **const handleFieldChange = (event) => {**  **const { name, value } = event.target;**  **if (name === "userweight") setUserWeight(value);**  **};**  **const handleSubmit = (event) => {** |

7. 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();** |

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

|  |
| --- |
| **const handleSubmit = (event) => {**  **event.preventDefault();**  **setError("");**  **if (userweight === "") {**  **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.

9. 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.

10. 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 where the other definitions are but 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, but that is an advanced topic.

11. 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. Aso the payload is already an object, so no need for curly braces.

1. 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.

1. 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(() => {** |

Notice that we end up with three then() methods!

1. 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.

Remember the db.json now has a completely new section for weights.

1. There is a problem with the code above, if the user is NOT logged in, navigating to the enter weight link causes the program to crash. The payload object is being created at the component level and it's trying to access user.id even when user might be null or undefined when the user is not logged in. We need to move the payload creation into the handleSubmit() function or we can check to see if the user exists before accessing it’s properties:

|  |
| --- |
| **const handleSubmit = (event) => {**  **event.preventDefault();**  **setError("");**  **const payload = {**  **id: user.id,**  **weight: String(userweight),**  **date: formattedDate**  **};**  **if (userweight === "") {** |

Of course it would be a good idea to add a button in the enter weight view to re-direct user to login view (if they are not logged in).

# part 03 – 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";** |

1. 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.

1. 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.

1. Here is the final function being cached:

|  |
| --- |
| **const handleCancelLogout = useCallback(() => {**  **setShowPopup(false);**  **}, []);** |

An empty array means no dependencies

1. 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.

1. Now, just before the return from Main, 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, an inner 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.

1. 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, so something in allEmployees.

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.

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

|  |
| --- |
| **sortedEmployees.map((customer, i) =>**  **(<div key={i}>**  **{customer.username}&nbsp;** |

1. To test this new feature try moving around the objects in the employees array. When you now go to the employees route,you will see they are re-arranged alphabetically for you based on the code in #5 above.

# part 04 – Modernizing with Async/Await

Using the Promise chain is absolutely not incorrect, but refactoring with async/await can certainly shorten the code and make it more readable. We will refactor the code in the main.jsx file of the Enterweight component with this more modern style.

For this file we can replace the Promise chain with async/await, place all the code into a try/catch block, simplify error handling and streamline the Response parsing.

1. First we change the handleSubmit() function like this:

|  |
| --- |
| **if (name === "userweight") setUserWeight(value);**  **};**  **const handleSubmit = async (event) => {**  **event.preventDefault();**  **setError("");** |

So now, the entire handleSubmit function is an asynchronous one. This will cause syntax errors below, but we will fix soon.

1. The area where the fetch() event occurs and the consequent code will need to be wrapped inside of a try/catch block. The fetch() has three then() methods and one catch() method associated with it:

|  |
| --- |
| **try {**  **fetch('http://localhost:3030/weights', {**  **method: 'POST',**  **headers: {**  **'Accept': 'application/json',**  **…other code goes here…**  **.then(() => {**  **navigate("/");**  **})**  **.catch(err => {**  **console.log("An error occured! " + err.message);**  **setIsSubmitting(false);**  **});**  **} catch (err) {**  **}** |

Since there is so much code in this section, I did not show all the code but you should be able to see where the try starts and where the catch ends the section.

1. Lets deal with the errors first, so the logic that was in the current cath() function now has to go into the catch() section of the try block:

|  |
| --- |
| **navigate("/");**  **} catch (err) {**  **console.log("An error occured! " + err.message);**  **setIsSubmitting(false);**  **}** |

1. With async there must be an await, this would go whereeve we are expecting some kind of blocking code, like when the fetch() executes. The arrival of that data is out of our program’s space so it could be a while, so we *await* the response:

|  |
| --- |
| **setIsSubmitting(true);**  **try {**  **const response = await fetch('http://localhost:3030/weights', {**  **method: 'POST',**  **headers: {** |

With this style, we creat a new function property to hold the data once it returns from the await() execution.

1. We can now handle the response part, not using the then() method:

|  |
| --- |
| **body: JSON.stringify(payload)**  **});**  **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}`);**  **}** |

Notice we don’t return the text() part of the response, see below.

1. With this new code style, we just apply the text() method to the original response we got after firing the fetch() method:

|  |
| --- |
| **throw new Error(`Server error: ${response.status}`);**  **}**  **const text = await response.text();** |

1. We can do a quick test for errors in the text we got from #6:

|  |
| --- |
| **const text = await response.text();**  **if (text) {**  **JSON.parse(text);**  **}**  **navigate("/");** |

By parsing the text object in this way, if any errors in the JSON, we will throw an automatic error. If all is well, just navigate out of here.

Hopefully you can see how much simpler this style of code is.

Test the app to see that it functions as before.

1. (Optional) If you want you can shorten the date to just one line:

|  |
| --- |
| **const { user } = useContext(AuthContext);**  **const formattedDate = new Date().toLocaleDateString('en-GB');**  **const handleFieldChange = (event) => {** |

Here we fire the new Date() function and immediately apply a format

1. (Optional) If you want you can shorten the date to just one line:

|  |
| --- |
| **const { user } = useContext(AuthContext);**  **const formattedDate = new Date().toLocaleDateString('en-GB');**  **const handleFieldChange = (event) => {** |

Here we fire the new Date() function and immediately apply a format

1. (Optional) Another change to stay modern, we could change the userweight field on the HTML side:

|  |
| --- |
| **<form onSubmit={handleSubmit}>**  **<div>**  **<label>**  **Today’s weight:**  **<input**  **type="number"**  **name="userweight"**  **min="20"**  **max="250"**  **step="1"**  **inputMode="decimal"**  **value={userweight ?? ''}**  **onChange={(e) => setUserWeight(e.target.valueAsNumber)}**  **onWheel={(e) => e.target.blur()}**  **disabled={isSubmitting}**  **required**  **/>**  **</label** |

Notice that the onChange event is changed, the function property can be updated right here in the field. We are finally taking advantage of the isSubmitting property. This means that if the form is being submitted, the field is disabled, nothing can be entered in here. The identifier onWheel in a React component is a JSX event prop that wires a wheel (scroll wheel / trackpad) listener to the underlying DOM element. When the user scrolls over that element using a mouse wheel, trackpad gesture, or similar pointing device, React invokes the handler, which in this case is to move away from this field. This also prevents accidental scroll value changes.

The prop value={userweight ?? ''} makes the input a controlled component whose displayed value is either the numeric state (userweight) or an empty string if the state is null or undefined. This means that a weight of 0 will not be treated as an error if the || operator was used instead.

The call setUserWeight(e.target.valueAsNumber) updates React state  
with the numeric value from the number input.   
valueAsNumber is a DOM property provided for inputs of type="number" (and range); the browser parses the current textual value and returns a JavaScript number.

1. (Optional) If you did the changes from #10, then you must change how errors are handled. Move the numeric checks to above the payload code:

|  |
| --- |
| **setError("");**  **if (userweight === null || Number.isNaN(userweight)) {**  **setError("Weight is required.");**  **return;**  **}**  **if (userweight < 20 || userweight > 250) {**  **setError("Weight must be between 20 and 250.");**  **return;**  **}**  **setIsSubmitting(true);**  **const payload = {** |

1. (Optional) Now we can use userweight as a number:

|  |
| --- |
| **setIsSubmitting(true);**  **const payload = {**  **id: user.id,**  **weight: userweight, // send as number**  **date: formattedDate**  **};**  **try {** |

Note: you will see two versions of this part in the downloaded zipped files. We will be proceeding with the state of the app with all the optional parts.

# part 05 – Preparing for Child Routes

We will demonstrate child routes by splitting the login process into two separate logins. One for the existing login we have now and one for future admin login. The CSS file will obviously change but the changes are done and included in the file for Part05.

1. First we re-arrange the login display. Here we can code the list item tag in a more vertical manner. The split the elements inside of the <li> tags into the original NavLink to and a new <div>:

|  |
| --- |
| **<li><NavLink to="/enterweight">enter weight</NavLink></li>**  **{ !user? (**  **<li>  <NavLink to="/login">login &#x25BE;</NavLink>  <div></div>  </li>**  **):(**  **<li onClick={handleLogoutClick}>{user.username}</li>**  **)}** |

Notice the *Hexadecimal Numeric Reference* for the downwards black arrow. It is the the hexadecimal code point 25BE.

1. Inside of the <div> tags add two more NavLink to tags:

|  |
| --- |
| **{ !user? (**  **<li>  <NavLink to="/login">login</NavLink>  <div>**  **<NavLink to=" " ><NavLink>**  **<NavLink to=" " ><NavLink>**  **</div>  </li>**  **):(**  **<li onClick={handleLogoutClick}>{user.username}</li>**  **)}** |

1. Now identify where these links are point to:

|  |
| --- |
| **{ !user ? (**  **<li >**  **<NavLink to="/login****" >login &#x25BE;</NavLink>**  **<div >**  **<NavLink to="/login">User Login</NavLink>**  **<NavLink to="/admin">Admin Login</NavLink>**  **</div>**  **</li>**  **) : (** |

1. Add in the CSS classes to make the effort more smooth:

|  |
| --- |
| **{ !user ? (**  **<li className="has-dropdown" >**  **<NavLink to="/login" >login &#x25BE;</NavLink>**  **<div className="nav-dropdown" >**  **<NavLink to="/login">User Login</NavLink>**  **<NavLink to="/admin" className="admin-option">Admin Login</NavLink>**  **</div>**  **</li>**  **) : (** |

Remember all the CSS classes have already been written for you and are in the styles.css file for this part and moving forward. So these styles would NOT have existed for earlier parts of the bootcamp files.

1. Before we move to coding this new route, lets add a commented path for Admin in the App.jsx file:

|  |
| --- |
| **<Route path='/register' element={<Register />} />**  **<Route path="/login" element={<Login />} />**  **<Route path='/enterweight' element={<Enterweight />} />**  **{/\* <Route path='/admin' element={<Admin />} /> \*/}**  **</Routes>**  **</BrowserRouter>** |

We will change this in part 06.

# part 06 – Child Routes

A screen shot of a computer screen

AI-generated content may be incorrect.We will now create a blank admin view so that we can protect that route and view. Copy any of the other components and rename accordingly. For example I copied the Home component and renamed it to the Admin component.

1. After copying the Home component and renaming the usual files we can now complete the path in App.jsx:

|  |
| --- |
| **import Enterweight from "./components/enterweight/enterweight";**  **import Admin from "./components/admin/admin";**  **function App() {** |

We already created a path for it in Part 05, you can now uncomment that line

1. The admin view will be for calculating the winner, so lets create yet another component called AdminLogin just for the purposes of logging in the administrator. Repeat the steps you did for the normal Admin component but call this new component AdminLogin:

|  |
| --- |
|  |

1. Since the admin route must be protected, lets create a ProtectedAdminRoute.

|  |
| --- |
|  |

This will be just a single .jsx file but it will use the existing AuthContext for user details. The purpose of this component is to protect the final Admin view.

1. We can work on the ProtectedAdminRoute now, here is the skeleton of that file:

|  |
| --- |
| **import { useContext } from 'react';**  **import { Navigate } from 'react-router-dom';**  **import { AuthContext } from './AuthContext';**  **function ProtectedAdminRoute( ) {**    **};**  **export default ProtectedAdminRoute;** |

We need useContext hook here so that we can access objects in a parent component. Here we want to access the current value of the React Context named AuthContext where we can get data about the current user.

1. Lets use the useContext hook to access the user from AuthContext. First we need to access the children of the caller component. This ProtectedAdminRoute will be called from App.jsx. During that call, App will provide an inner child to the ProtectedAdminRoute component. We need a mechanism to allow this communication, React uses a parameter called children:

|  |
| --- |
| **import { useContext } from 'react';**  **import { Navigate } from 'react-router-dom';**  **import { AuthContext } from './AuthContext';**  **function ProtectedAdminRoute({ children }) {**  **};**  **export default ProtectedAdminRoute;** |

The children parameter in this case is a Prop. So ProtectedAdminRoute in this case is a **Higher Order Component** that needs access to a child component, Admin in this case. The children component will remain hidden until we apply logic to see who is logging in. If an admin is logging in, we reveal the child.

1. Now we can destructure user from AuthContext:

|  |
| --- |
| **import { useContext } from 'react';**  **import { Navigate } from 'react-router-dom';**  **import { AuthContext } from './AuthContext';**  **function ProtectedAdminRoute({ children }) {**  **const { user } = useContext(AuthContext);**  **};**  **export default ProtectedAdminRoute;** |

1. Users will eventually have roles, so we can add logic here to determine what kind of user we have:

|  |
| --- |
| **import { useContext } from 'react';**  **import { Navigate } from 'react-router-dom';**  **import { AuthContext } from './AuthContext';**  **function ProtectedAdminRoute({ children }) {**  **const { user } = useContext(AuthContext);**  **if (!user || user.role !== 'admin') {**  **return <Navigate to="/admin-login" replace />;**  **}**  **};**  **export default ProtectedAdminRoute;** |

1. Now that the user is authenticated and has the 'admin' role, the children component is rendered to the screen:

|  |
| --- |
| **import { useContext } from 'react';**  **import { Navigate } from 'react-router-dom';**  **import { AuthContext } from './AuthContext';**  **function ProtectedAdminRoute({ children }) {**  **const { user } = useContext(AuthContext);**  **if (!user || user.role !== 'admin') {**  **return <Navigate to="/admin-login" replace />;**  **}**  **return children;**  **};**  **export default ProtectedAdminRoute;** |

1. Moving over to the App.jsx file, we can now code in an additional route for admin-login to open the AdminLogin logic.

|  |
| --- |
| **<Route path='/register' element={<Register />} />**  **<Route path="/login" element={<Login />} />**  **<Route path='/enterweight' element={<Enterweight />} />**  **{/\* <Route path='/admin' element={<Admin />} /> \*/}  <Route path='/admin-login' element={<AdminLogin />}**  **</Routes>**  **</BrowserRouter>** |

1. The commented /admin path now will be protected by diverting the user to the ProtectedAdminRoute:

|  |
| --- |
| **<Route path='/register' element={<Register />} />**  **<Route path="/login" element={<Login />} />**  **<Route path='/enterweight' element={<Enterweight />} />**  **<Route path='/admin' element={<ProtectedAdminRoute><Admin /></ProtectedAdminRoute>} /> <Route path='/admin-login' element={<AdminLogin />}**  **</Routes>**  **</BrowserRouter>** |

1. We should update the Header component to make sure we understand what each link does and where they lead to:

|  |
| --- |
| **<Route path="/login" element={<Login />} />**  **{ !user ? (**  **<li className="has-dropdown" >**  **<NavLink to="/login" >login &#x25BE;</NavLink>**  **<div className="nav-dropdown" >**  **<NavLink to="/login">User Login</NavLink>**  **<NavLink to="/admin-login" className="admin-option">Admin Login</NavLink>**  **</div>**  **</li>**  **) : (**  **<>**  **{user.role === 'admin' && <li><NavLink to="/admin">Admin Panel</NavLink></li>}**  **<li onClick={handleLogoutClick}>{user.username}</li>**  **</>**  **) }**  **</ul>** |

Add a separate section where we can now access the role of the user. If the user is an admin, then allow them to access the /admin path, which is now protected. We will expand the db.json file to add the admin part soon.

# part 07 – Admin Login Logic

We created the new AdminLogin component but we still have to validate the user. We need to modify the db.json file for the new role feature and then complete the admin login form and logic. In this part we will be working mainly in the main.jsx file for the AdminLogic component.

1. In the Main component of AdminLogin, import these hooks and libraries:

|  |
| --- |
| **import { useState, useContext } from "react";**  **import { useNavigate } from "react-router-dom";**  **import { AuthContext } from "./../AuthContext";**  **function Main() {** |

We have used all of these hooks throughout the bootcamp.

1. Create the state properties or variables. Of course we need the username and password, but we also need to get access to the login() method on the AuthContext and be able to navigate to /admin, the actual goal:

|  |
| --- |
| **function Main() {**  **const [username, setUsername] = useState("");**  **const [password, setPassword] = useState("");**  **const navigate = useNavigate();**  **const { login } = useContext(AuthContext);** |

1. Since this is the UI part of the login process, we need to handle the username and password being entered as well as the form submi event:

|  |
| --- |
| **const handleFieldChange = (event) => {**  **const { name, value } = event.target;**  **};**  **const handleSubmit = (event) => {**  **event.preventDefault();**  **};**  **return (** |

We know the form controls (fields) will return an event object based on which control raised the event, so just add that in now. We also know that the form submit event needs the preventDefault() function, so add that in now

1. On day 1 we added the AuthContext and that had a login() function. We need to call that function with the valid user once the fetch() is successful:

|  |
| --- |
| **import { useContext } from 'react';**  **import { Navigate } from 'react-router-dom';**  **import { AuthContext } from './AuthContext';**  **function ProtectedAdminRoute( ) {**    **};**  **export default ProtectedAdminRoute;** |

We need useContext hook here so that we can access objects in a parent component. Here we want to access the current value of the React Context named AuthContext where we can get data about the current user.

1. Lets fire the fetch() method and handle the JSON response:

|  |
| --- |
| **const handleSubmit = (event) => {**  **event.preventDefault();**  **fetch('http://localhost:3030/employees')**  **.then(response => response.json())** |

1. Now we know we will get a user object, lets create a local variable here to store that user object:

|  |
| --- |
| **fetch('http://localhost:3030/employees')**  **.then(response => response.json())**  **.then(users => {**  **const found = users.find(**  **user => user.username === username && user.password === password && user.role === 'admin'**  **);** |

Notice that we now have the role being passed. We still need to change the db.json file to accommodate this new property of the user object, soon.

1. After #6, we may or may not have a valid admin user. Lets proceed as though we will always get a valid user, we will handle the “not found” part later:

|  |
| --- |
| **.then(users => {**  **const found = users.find(**  **user => user.username === username && user.password === password && user.role === 'admin'**  **);**  **if(found){**  **login(found);**  **navigate("/admin");**  **}**  **})** |

It is here that we call that login() function on AuthCOntext and pass it the user object. At the same time we can allow the browser to the /admin path

1. Lets do a general error catch() at the end of this handleSubmit():

|  |
| --- |
| **if(found){**  **login(found);**  **navigate("/admin");**  **} else {**  **};**  **})**  **.catch( err => console.error("An error occured!" + err.message));**  **};**  **return (** |

1. Now we should keep track of the status of this login process. If the login was NOT successful we update the status and move on:

|  |
| --- |
| **function Main() {**  **const [username, setUsername] = useState("");**  **const [password, setPassword] = useState("");**  **const [loginStatus, setLoginStatus] = useState("");**  **const navigate = useNavigate();**  **const { login } = useContext(AuthContext);** |

Add the loginStatus state variable where the other declarations are.

1. After #6, we may or may not have a valid admin user. Lets proceed as though we will always get a valid user, we will handle the “not found” part later:

|  |
| --- |
| **const found = users.find(**  **user => user.username === username && user.password === password && user.role === 'admin'**  **);**  **if(found){**  **setLoginStatus("Admin Login Successful");**  **login(found);**  **navigate("/admin");**  **} else {**  **setLoginStatus("Invalid admin credentials");**  **}** |

We also add an else{} clause to cater for invalid login via the login status.

1. We could also chage the catch to update the login status (you could leave the console log if you want):

|  |
| --- |
| **} else {**  **setLoginStatus("Invalid admin credentials");**  **}**  **})**  **.catch( err => setLoginStatus("An error occured!" + err.message));**  **};** |

1. If we have the status, well we coud display that status at the bottom of the form, on the view:

|  |
| --- |
| **} else {**  **setLoginStatus("Invalid admin credentials");**  **}**  **})**  **.catch( err => setLoginStatus("An error occured!" + err.message));**  **};** |

1. Finally we can handle the db.json file. It is very easy, just create one more key/value pair and just make some object just a user object and choose one or more to be admins:

|  |  |
| --- | --- |
| **"employees": [**  **{**  **"username": "Axle",**  **"password": "1234",**  **"id": "a4fb",**  **"role": "admin"**  **},** | **{**  **"id": "a97e",**  **"username": "Jane",**  **"password": "1234",**  **"role": "user"**  **},** |

Optional features for Part 07

If you wanted to put the focus back to the username field, add the following hooks

1. We could also chage the catch to update the login status (you could leave the console log if you want):

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

1. Add a reference to the useRef() hook, which always start out as null:

|  |
| --- |
| **const navigate = useNavigate();**  **const { login } = useContext(AuthContext);**  **const focusUser = useRef(null);** |

1. Add a direct reference to the actual DOM element, the username input field:

|  |
| --- |
| **<label>**  **Name:**  **<input**  **type="text"**  **name="username"**  **onChange={handleFieldChange}**  **ref={focusUser}**  **/>**  **</label>** |

Remember that this allows the component to directly access and manipulate the input element

1. Remember also that if we tried to access the usernamem field before it exists, we will get an error. We need to wrap the focus() event inside of a useEffect() hook which makes the focus() event run as a side effect, just once:

|  |
| --- |
| **const focusUser = useRef(null);**  **useEffect(() => {**  **focusUser.current.focus();**  **}, []);**  **const handleFieldChange = (event) => {** |