Table of Contents

[**part 01 – Component Organization** 2](#_Toc34920444)

[**part 02 – Authorization and Authentication** 5](#_Toc34920445)

[**part 03 – Adding Authorization Middleware** 7](#_Toc34920446)

[**part 04 – Logging In** 12](#_Toc34920447)

[**part 05 – Adding a Weight** 20](#_Toc34920448)

[**part 06 – Reading The JWT** 23](#_Toc34920449)

[**part 07 – Adding Links to Menu Items** 26](#_Toc34920450)

[**part 08 – Fixing team weights** 28](#_Toc34920451)

[**part 09 – Fixing team weights** 29](#_Toc34920452)

[**part 10 – Adding a new Employee (API)** 32](#_Toc34920453)

[**part 11 – Adding a new Employee (Login Status)** 34](#_Toc34920454)

[**part 12 – Adding a New Search API (Optional)** 36](#_Toc34920455)

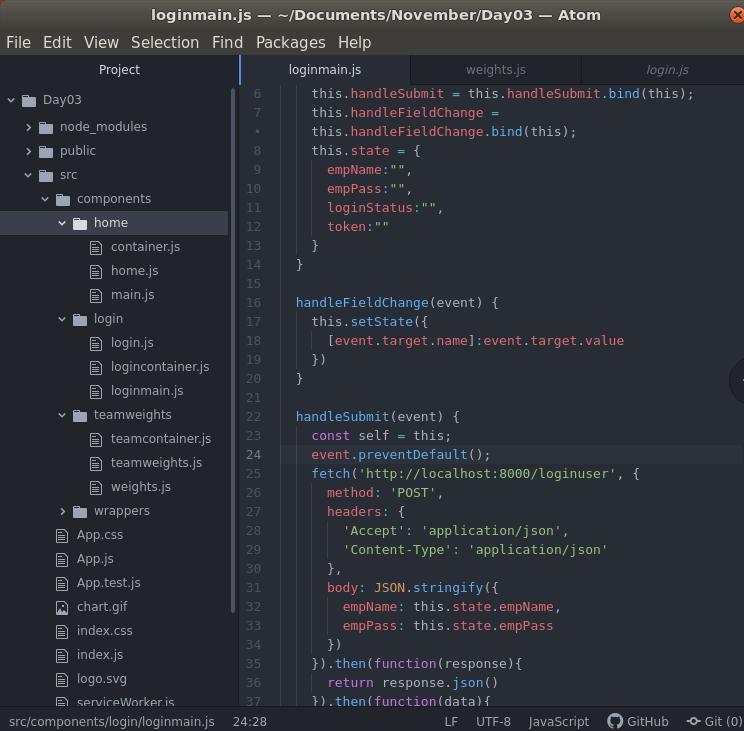
[**part 13 – Implement Search API in ReactJS PartA (Optional)** 37](#_Toc34920456)

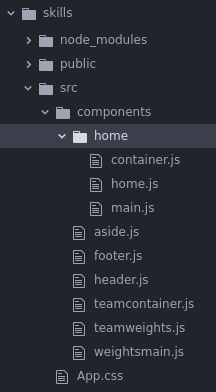
[**part 14 – Implement Search API in ReactJS PartB (Optional)** 41](#_Toc34920457)

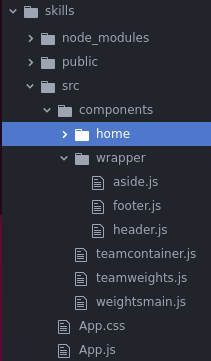
# part 01 – Component Organization

Day04 continues directly from Day03. You may choose to rename your folder or just keep working your Day03 folder.

1. Start separating out our components into home, enter weight and the rest of the menu links. We would also need views for logging in/out and admin.



1. If you look at the menu on the site, you will see home, enter weight, my weight and team weights. We will now arrange our folder structure inside of components to reflect this menu
2. Starting with home, create a folder inside of components called *home* and put all the .js files that you think belong to the *home* ‘view’.
3. Put all the wrapper files like aside footer and header into a folder called wrapper



1. Of course all the links will break, so lets fix those one by one. Work on the home folder first then replicate this success to the other folders. Take a look at home.js inside of the home folder, change the paths to reflect our changes.

|  |
| --- |
| **import React from "react";**  **import Container from "./container";**  **import Footer from "../wrapper/footer";**  **import Header from "../wrapper/header";**  **//**  **function Home(){** |

1. In the container.js file inside of the home folder, change the aside’s **path**

|  |
| --- |
| **import React from "react";**  **import Main from "./main";**  **import Aside from "../wrapper/aside";**  **function Container(){** |

1. Now of course our App.js has to change to reflect the new location of **home**, so in App.js change line 3 or wherever **home** is being imported.

|  |
| --- |
| **import React from "react";**  **import Main from "./main";**  **import Aside from "../wrapper/aside";**  **function Container(){** |

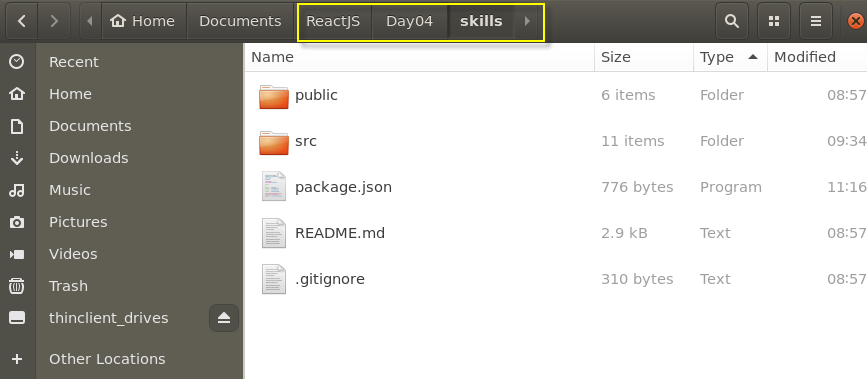
1. In header.js fix our logo path

|  |
| --- |
| **import React from "react";**  **import logo from '../../chart.gif';**  **//**  **function Header(){** |

1. Now repeat steps 1-8 for teamweights.

As this task is very repititive, we will provide you with the entire application at this point. So download skills.zip and install it by running npm install.

You can install this app into a folder called *skills*, inside of your Day04 folder. You may have to rename the zipped folder to *skills*.



**This is before I run npm install.**

Note also that a couple of minor warnings were fixed in this zip file

# part 02 – Authorization and Authentication

Authorization can take place within the React app or externally to the API or both. We will look at authentication against the NodeJS API in this course. Our authorization framework will involve tokens.

1. Kill the NodeJS application with CTRL+C, then run the following command to install JWT

Note: the NodeJS app may still be running in the Day02 folder, you could leave it there or move it to Day04 and rename it to something like HTTPServer

**npm install jsonwebtoken**

You can restart the application using **npm run dev**

1. In controller.js file import the **jsonwebtoken**

|  |
| --- |
| **const schema = require('./validate');**  **const Employee = require('../models/employees');**  **const jwt = require('jsonwebtoken');**  **//**  **exports.getdefault = function(request, response){**  **response.send('You are on the root route.');** |

1. In controller.js file, create a function called **loginuser()** with boilerplate code. We know we need the users **empName** and **empPass** in order to log in.

|  |
| --- |
| **const Employee = require('../models/employees');**  **const jwt = require('jsonwebtoken');**  **exports.loginuser = function(request, response){**  **let empName = request.body.empName;**  **let empPass = request.body.empPass;**  **};**  **exports.getdefault = function(request, response){** |

1. Now implement the **find()** function to find the user seeking access (or a token in this case). We already know that if an error occurred, we will send an appropriate message, but for now lets just send the entire **error** object.

|  |
| --- |
| **exports.loginuser = function(request, response){**  **let empName = request.body.empName;**  **let empPass = request.body.empPass;**  **Employee.find({empName:empName}, function(err, results) {**  **if (err) response.end(err);**  **});**  **};** |

1. If we find the user, then we proceed otherwise we send back to the client that the login failed. Notice I am just checking the passwords here (no encryption yet)

|  |
| --- |
| **Employee.find({empName:empName}, function(err, results) {**  **if (err) response.end(err);**  **if (empPass == results[0].empPass){**  **} else {**  **response.send({status:"Login Failed"});**  **}**  **});** |

1. If the passwords match then we have a valid user, we can use the **jwt** object to generate a token and send it back to the requesting client. Notice the **sign()** method of the JWT takes at the minimum 4 parameters, two of which are objects themselves

|  |
| --- |
| **if (empPass == results[0].empPass){**  **jwt.sign({ },**  **"",**  **{},**  **function(){ }**  **)**  **} else {**  **response.send({status:"Login Failed"});**  **}** |

The **sign()** method takes several parameters. Once **find()** from MongoDB returns, it will contain the name and password, so we extract that using the array syntax and passing that as a payload to the **sign()** method of **jwt**. We also pass a piece of text (any text) which will act as our *key*, then an expiry date and finally a function that asynchronously returns the token to the client.

1. Here is the complete **sign()** method

|  |
| --- |
| **if (empPass == results[0].empPass){**  **jwt.sign({**  **empName:results[0].empName,**  **userID:results[0].\_id**  **},**  **"mysecret",**  **{expiresIn : "1h"},**  **function(err, token){**  **if(err) throw err;**  **response.json({token:token});**  **}**  **)**  **} else {**  **response.send({status:"Login Failed"});**  **}** |

1. Here is the entire **loginuser()** function

|  |
| --- |
| **exports.loginuser = function(request, response){**  **let empName = request.body.empName;**  **let empPass = request.body.empPass;**  **Employee.find({empName:empName}, function(err, results) {**  **if (err) response.end(err);**  **if (empPass == results[0].empPass){**  **jwt.sign({**  **empName:results[0].empName,**  **userID:results[0].\_id**  **},**  **"mysecret",**  **{expiresIn : "1h"},**  **function(err, token){**  **if(err) throw err;**  **response.json({token:token});**  **}**  **)**  **} else {**  **response.send({status:"Login Failed"});**  **}**  **});**  **};** |

# 

# part 03 – Adding Authorization Middleware

1. We now need to add middleware to the call stack in order to verify the token being passed by a user. Back in the NodeJS application, in the controllers folder add a new .js file called auth.js, then start with the following boilerplate code:

|  |
| --- |
| **const jwt = require("jsonwebtoken");**  **module.exports = function (req, res, next) {**  **}** |

1. Although tokens can be sent in several ways, it is conventional to send them via the *headers* part of a request. In this case the token is simply being sent via the response object. Lets create a new variable to hold that token value from the authorization headers.

|  |
| --- |
| **const jwt = require("jsonwebtoken");**  **module.exports = function (req, res, next) {**  **const rawToken = req.headers.authorization;**  **}** |

1. We can now use the **jwt** object to verify the token we just got from the headers section of the request

|  |
| --- |
| **const jwt = require("jsonwebtoken");**  **module.exports = function (req, res, next) {**  **const rawToken = req.headers.authorization;**  **const decToken = jwt.verify(rawToken, 'mysecret');**  **}** |

Notice that we also have to pass the *key* to the verify function as the second parameter.

We can now store the decToken in a response object ready for sending back to the client. (we wont actually be sending anything back, just verifying the token here)

1. This code as it is will not work, the authorization header contains some extra information by convention, it has the word “Bearer” then a space then the actual token, we need to extract only the token, so the split function will work nicely.

|  |
| --- |
| **const jwt = require("jsonwebtoken");**  **module.exports = function (req, res, next) {**  **const rawToken = req.headers.authorization.split(" ")[1];**  **const decToken = jwt.verify(rawToken, 'mysecret');**  **}**  **}** |

1. Now we just call the next() function in the call stack.

|  |
| --- |
| **const jwt = require("jsonwebtoken");**  **module.exports = function (req, res, next) {**  **const rawToken = req.headers.authorization.split(" ")[1];**  **const decToken = jwt.verify(rawToken, 'mysecret');**  **next();**  **}**  **}** |

1. The JWT does not have native error handlers, so wrap up the code in try catch block for safety

|  |
| --- |
| **const jwt = require("jsonwebtoken");**  **module.exports = function (req, res, next) {**  **try{**  **const rawToken = req.headers.authorization.split(" ")[1];**  **const decToken = jwt.verify(rawToken, 'mysecret');**  **req.userInfo = decToken;**  **next();**  **}catch(error){**  **return res.status(401).json({message:"not authorized"});**  **}**  **}** |

1. All that’s left now is to protect a route, first import the auth.js file we just created into the routes.js file

|  |
| --- |
| **const bodyParser = require('body-parser');**  **const controller = require('../controllers/controller');**  **const auth = require('../controllers/auth');**  **module.exports = function(app){**  **app.use(bodyParser.urlencoded({extended:false}));** |

1. We will use **getweights** route to implement our authorizatoin code. Simply insert the *auth* variable before the controller part

|  |
| --- |
| **app.route('/').get(controller.getdefault);**  **app.route('/addnewemployee').post(controller.addnewemployee);**  **app.route('/getalldocs').get(auth, controller.getalldocs);**  **app.route('/addnewweight').put(controller.addnewweight);**  **app.route('/loginuser').post(controller.loginuser);** |

1. If you use the browser, you should get a ReactJS error as we are attempting to parse an empty array, so we would need to add some error handling in the weightsmain.js file and try again. Wrap the entire **return()** method into an **if** structure

|  |
| --- |
| **render(){**  **if(this.state.allWeights.length > 0){**  **return(**  **<main>**  **{this.state.allWeights.map((emp, i) =>** …  **)**  **)}**  **</main>**  **)**  **} else {**  **return(**  **<div>Not authorized</div>**  **)**  **}**  **}** |

1. Note the render function within weightsmain.js is long and complicated, please refer to the box below for the entire function

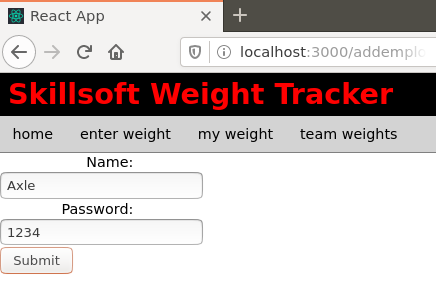
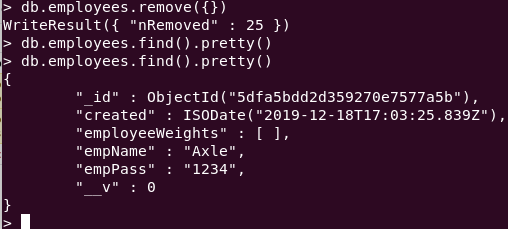
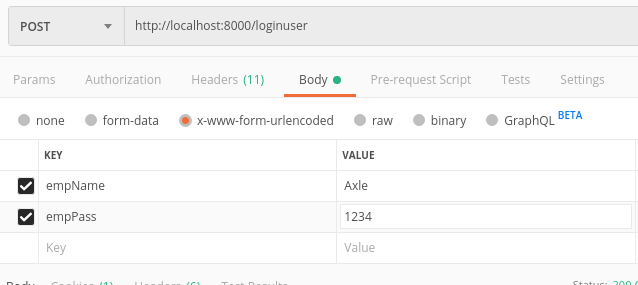
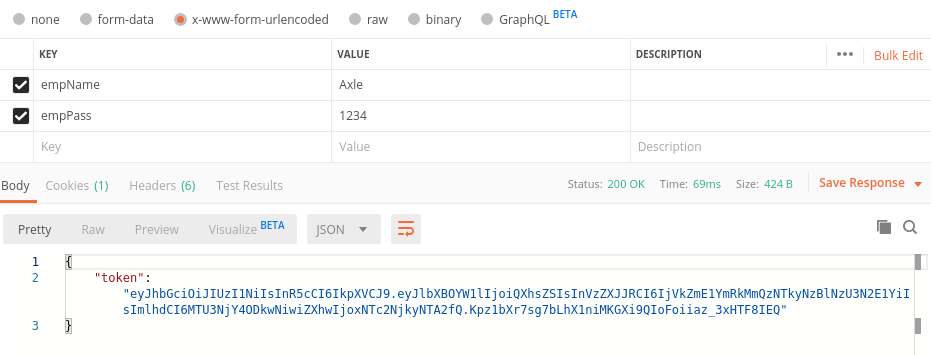
|  |
| --- |
| **render(){**  **if(this.state.allWeights.length > 0){**  **return(**  **<main>**  **{this.state.allWeights.map((emp, i) =>**  **(**  **<div key={i}>**  **{ this.state.allWeights.map((emp,j) =>**  **(**  **<div key={j}>**  **{emp.empName}{**  **emp.employeeWeights.map((weight,k)=>{**  **return <div key={k}>**  **Date: {new Date(weight.weighedDate).toLocaleDateString()}**  **{' '}**  **Weight: {weight.empWeight}**  **</div>**  **})**  **}**  **</div>**  **)**  **)**  **}**  **</div>**  **)**  **)}**  **</main>**  **)**  **} else {**  **return(**  **<div>Not authorized</div>**  **)**  **}**  **}** |

1. Now try to go to that endpoint either in the browser or with Postman  
   

Notice it says *Not authorized*

1. In routes.js file, add a POST route to handle user login, now that we have the controller function

|  |
| --- |
| **app.route('/addnewemployee').post(controller.addnewemployee);**  **app.route('/addnewweight').put(controller.addnewweight);**  **app.route('/loginuser').post(controller.loginuser);** |

1. First add a new user into the database using either the REST client or the browser. If you alread added a user from Part04 you may use that. In the example below I added a new user called Axle with password 1234 on Day02 Part06. You can see I verified that user using the database.
2. Now that we have a user, lets sign in that user to see if a token can be generated. The first step in this process is to use the REST client with the empName and empPass fields filled out, along with the url and restful
3. The second step is to hit send to generate the token, in my case this is what happened after hitting the *send* button on Postman: 

# part 04 – Logging In

We would need a login component in ReactJS to point to our login endpoint.

1. Duplicate the home folder and change the names of the .js files to reflect logging into the application. So main.js becomes loginmain.js, home.js becomes login.js and container.js becomes logincontainer.js

|  |
| --- |
|  |

1. Starting with loginmain.js, change the name of the function and the **<h2>** tags accordingly. Remove the three **<p>** tags.

|  |
| --- |
| **import React from "react";**  **function LoginMain(){**  **return (**  **<main>**  **<h2>Login</h2>**  **</main>**  **)**  **};**  **export default LoginMain** |

1. In logincontainer.js, import **LoginMain** instead of **Main** and change the name of the container component

|  |
| --- |
| **import React from "react";**  **import LoginMain from "./loginmain";**  **import Aside from "./../wrapper/aside";**  **function LoginContainer(){**  **return (**  **<div>**  **<LoginMain />**  **<Aside />**  **</div>**  **)**  **}** |

4. In login.js change the code accordingly

|  |
| --- |
| **import React from "react";**  **import LoginContainer from "./logincontainer";**  **import Footer from "./../wrapper/footer";**  **import Header from "./../wrapper/header";**  **//**  **function Login(){**  **return (**  **<div>**  **<Header />**  **<LoginContainer />**  **<Footer />**  **</div>**  **)**  **}**  **//**  **export default Login** |

5. Add this view to the routing component in App.js and test.

|  |
| --- |
| **mport './styles.css';**  **import Home from './components/home/home';**  **import Login from './components/login/login';**  **…**  **<BrowserRouter>**  **<Route exact path="/" component={Home}/>**  **<Route path="/teamweights" component={TeamWeights}/>**  **<Route path="/login" component={Login}/>**  **</BrowserRouter>** |

|  |
| --- |
|  |

6. Now we can add a form that will accept the name and password. We first have to turn our function in loginmain.js into a class so that we can use **state**. Import **component** from react and change the function into a class.

|  |
| --- |
| **import React, {Component} from 'react';**  **class LoginMain extends Component{**  **return (**  **<main>**  **<h2>Login</h2>**  **</main>**  **)**  **};**  **//**  **export default LoginMain** |

7. You should immediately get an error because classes *render* HTML. Add a **render()** function to wrap the **return** statement.

|  |
| --- |
| **class LoginMain extends Component{**  **render() {**  **return (**  **<main>**  **<h2>Login</h2>**  **</main>**  **);**  **}**  **};** |

8. In order to use ReactJS **state**, we need a constructor in this class that references the **props** property via a call to **super().**

|  |
| --- |
| **class LoginMain extends Component{**  **constructor(props) {**  **super(props);**  **}**  **render() {** |

9. At the same time we can define a few variables to hold state as we conduct busiiness

|  |
| --- |
| **class LoginMain extends Component{**  **constructor(props) {**  **super(props);  this.state = {**  **empName:"",**  **empWeight:"",**  **empPass:"",**  **loginStatus:"",**  **token:""**  **}**  **}**  **render() {** |

1. We have a structure for inserting a form, just add in the code below, but do it underneath the **<h2>** tag.

|  |
| --- |
| **<main>**  **<h2>Login</h2>**  **<form>**  **<div>**  **<label>**  **Name:**  **<input type="text" name="empName" />**  **</label>**  **</div>**  **<div>**  **<label>**  **Password:**  **<input type="text" name="empPass" />**  **</label>**  **</div>**  **<div>**  **<input type="submit" value="Submit" />**  **</div>**  **</form>**  **</main>** |

11. Lets add some functionality to the **onSubmit** event. We will define a function in this class called **handleSubmit()**

|  |
| --- |
| **<main>**  **<h2>Login</h2>**  **<form onSubmit={this.handleSubmit}>**  **<div>**  **<label> })**  **}**  **//** |

1. Add a function inside of the class which will invoke the **fetch()** method of JS. You can do this above the **render()** function. Also add in the **preventDefault()** call to avoid the form be submitted without values.

|  |
| --- |
| **}**  **}**  **handleSubmit(event) {  event.preventDefault();**  **fetch()**  **}**  **render() {**  **return (** |

1. With that in place, we need to register this new function with the *this* reference, in our constructor. If we did not do this, the *this* keyword would refer to the Window object and not our component. The *this* keyword in JS is relevant to the context in which it is used. We do this in the constructor.

|  |
| --- |
| **class LoginMain extends Component{**  **constructor(props) {**  **super(props);**  **this.handleSubmit = this.handleSubmit.bind(this);**  **this.state = {**  **empName:"",**  **empWeight:"",** |

1. At the same time, the *this* keyword will not be available in the **fetch()** function, we actually have to point **this** to another variable so that we can continue using it.

|  |
| --- |
| **handleSubmit(event) {**  **const self = this;**  **event.preventDefault();**  **fetch('http://localhost:8000/loginuser', {**  **method: 'POST',**  **headers: { },**  **body: JSON.stringify({ })**  **})**  **}** |

We can also insert what we know our **fetch()** mehod will need to complete it’s job.

1. We now complete the headers and body parameters.

|  |
| --- |
| **fetch('http://localhost:8000/loginuser', {**  **method: 'POST',**  **headers: {**  **'Accept': 'application/json',**  **'Content-Type': 'application/json'**  **},**  **body: JSON.stringify({**  **empName: this.state.empName,**  **empPass: this.state.empPass**  **})**  **})** |

1. When **fetch()** returns, in this case it should return with a token. We need to extract that token and store it in local storage for now. We will do this in two steps. Our **fetch()** method in my case, ends on line 28 and it is at that point we chain on a couple of **then()** methods and a **catch()** to handle the promise from our **fetch().**

|  |
| --- |
| **body: JSON.stringify({**  **empName: this.state.empName,**  **empPass: this.state.empPass**  **})**  **}).then().then().catch()**  **}** |

1. The first **then()** will simply return the server object in json format, from **fetch()**, after that the second **then()** will extract just the data portion of that server object.

|  |
| --- |
| **empName: this.state.empName,**  **empPass: this.state.empPass**  **})**  **}).then(function(response){**  **return response.json();**  **}).then(function(data){**  **//do something with data**  **}).catch()**  **}** |

1. Now that we have data we can assign it to our state object and log any errors if they occur.

|  |
| --- |
| **})**  **}).then(function(response){**  **return response.json();**  **}).then(function(data){**  **self.setState({token: data.token});**  **self.setState({loginStatus: 'Logged in'});**  **localStorage.setItem('token', self.state.token)**  **}).catch(function(err){**  **console.log(err);**  **})**  **}** |

1. Here is the entire handleSubmit() function

|  |
| --- |
| **handleSubmit(event) {**  **const self = this;**  **event.preventDefault();**  **fetch('http://localhost:8000/loginuser', {**  **method: 'POST',**  **headers: {**  **'Accept': 'application/json',**  **'Content-Type': 'application/json'**  **},**  **body: JSON.stringify({**  **empName: this.state.empName,**  **empPass: this.state.empPass**  **})**  **}).then(function(response){**  **return response.json();**  **}).then(function(data){**  **self.setState({token: data.token});**  **self.setState({loginStatus: 'Logged in'});**  **localStorage.setItem('token', self.state.token)**  **}).catch(function(err){**  **console.log(err);**  **})**  **}** |

1. Before we post anything, we need the values that the user types into the input boxes of the HTML form, so we do this with the *onchange* event. When that change happens we provide a function to handle that change, which really means storing the vaue inside the **state object**. So first lets write a class function to handle these values, you can insert it anywhere in the class.

|  |
| --- |
| **handleFieldChange(event) {**  **this.setState({**  **[event.target.name]:event.target.value**  **})**  **}** |

1. We will pass this function to the onChange event of the field. We capture the value using the event object. the word *event* can be anything, it can be just ‘e’. However with that object we can access both the name of the field and its value. Once we get the values and the field names, we invoke **setState()** to register these objects with our state object.
2. Just like with **handleSubmit**, we also have to register **handleFieldChange** in our constructor

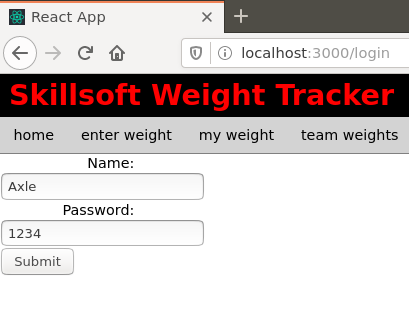
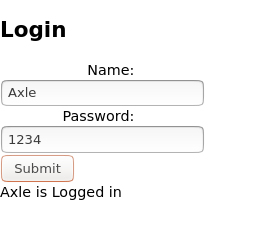
|  |
| --- |
| **class LoginMain extends Component{**  **constructor(props) {**  **super(props);**  **this.handleSubmit = this.handleSubmit.bind(this);**  **this.handleFieldChange = this.handleFieldChange.bind(this);**  **this.state = {**  **empName:"",**  **empWeight:"",** |

1. Now we can implement this function in the HTML

|  |
| --- |
| **<form onSubmit={this.handleSubmit}>**  **<div>**  **<label>**  **Name:**  **<input type="text" name="empName" onChange={this.handleFieldChange}/>**  **</label>**  **</div>**  **<div>**  **<label>**  **Password:**  **<input type="text" name="empPass" onChange={this.handleFieldChange}/>**  **</label>**  **</div>**  **<div>** |

1. Before we test the functionality, lets add some more HTML to display any feedback we get from our API.

|  |
| --- |
| **<div>**  **<input type="submit" value="Submit" />**  **</div>**  **<div>**  **<span>{this.state.empName} is {this.state.loginStatus}</span>**  **</div>**  **</form>**  **</main>** |

1. Now lets login using our */login* route and then try to */teamweights*  
     
   
2. If successful, you should see the message under the submit button   
   

Note, we will still get an unauthoried message if we try /teamweights, we need to do some more work on the weightsmain.js file

# part 05 – Adding a Weight

We need a form to extend our documents (add to them). Axle has a document started but he needs to record his weight each day, so for that we will use a different form. Notice we already know who is logged in, so we just have to use that same document and add to it. In other words, extend Axle’s document.

1. In Atom, we can duplicate the login component and rename everything to fit our new ‘enter weight’ form.

|  |
| --- |
|  |

1. Start with enterweightcontainer.js and edit the code to suite the new view

|  |
| --- |
| **import React from 'react';**  **import EnterWeightMain from "./enterweightmain";**  **import Aside from "./../wrappers/aside";**  **function EnterWeightContainer(){**  **return(**  **<div>**  **<EnterWeightMain />**  **<Aside />**  **</div>**  **)**  **}**  **export default EnterWeightContainer;** |

1. Then enterweight.js

|  |
| --- |
| **import React from "react";**  **import EnterWeightContainer from "./enterweightcontainer";**  **import Footer from "./../wrappers/footer";**  **import Header from "./../wrappers/header";**  **function EnterWeight(){**  **return (**  **<div>**  **<Header />**  **<EnterWeightContainer />**  **<Footer />**  **</div>**  **)**  **}**  **export default EnterWeight** |

1. This is just a quick class for now, we will change a few more things in the next step. I have collapsed the main functions using Atom collapsible feature

|  |
| --- |
|  |

1. Add a route in app.js so we can see this ‘view’

|  |
| --- |
| **import Login from './components/login/login';**  **import EnterWeight from './components/enterweight/enterweight';**  **import {BrowserRouter, Route} from 'react-router-dom';**  **function App() { …………….. component={Teamweights} />**  **<Route path="/login" component={Login} />**  **<Route path="/enterweight" component={EnterWeight} />**  **</BrowserRouter>** |

1. Then the field for weight in enterweightmain.js. there will only be one field for weight, because we already know who is logged in and the date. So after making the changes below remove the second *password* field. Also change the prompt at the top of the form from *Login* to *Enter Weight*.

|  |
| --- |
| **<main>**  **<h2>Enter Weight</h2>**  **<form onSubmit={this.handleSubmit}>**  **<div>**  **<span>**  **{this.state.empName}, what is your**  **</span>**  **</div>**  **<div>**  **<label>**  **Weight today:**  **<input type="text" name="empWeight" onChange={this.handleFieldChange}/>**  **</label>** |

1. But remember this form should reflect who is entering the weight, so lets insert a span tag with the name, above the field and label.

|  |
| --- |
| **render() {**  **return (**  **<main>**  **<form onSubmit={this.handleSubmit}>**  **<div>**  **<span>**  **{this.state.empName}, what is your**  **</span>**  **</div>**  **<div>**  **<label>**  **Weight today:**  **<input type="text" name="empWeight"** |

1. Make sure your state object looks like this one, we will fill in the details later

|  |
| --- |
| **this.handleSubmit = this.handleSubmit.bind(this);**  **this.handleFieldChange = this.handleFieldChange.bind(this);**  **this.state = {**  **empName:"",**  **empWeight:"",**  **loginStatus:"",  feedback:""**  **}** |

1. After logging in (use /login), try to render the enterweight view (/enterweight), you will see that the value of who is logged in does NOT show up. That’s because we have not really handled logging in and profile. Note that there are several ways to handle this. In the next section we will simply use the local storage cookie we created at login, **but this is NOT the best way to handle this**.

|  |
| --- |
|  |

# part 06 – Reading The JWT

In order for our app to know whose weight is being uploaded, we first need to find out who is logged in while in our ReactJS application.

1. If we are going to decode the token, we need the Json Web Token in our ReactJS application. So stop the app using CTRL-C and install the jwt in your **skills** folder just like you did for our NodeJS application, so: **npm install jsonwebtoken**
2. Remember to install JWT and require the JWT at the top of the class where you will be using this feature, so enterweightmain.js in this case.

|  |
| --- |
| **import React, {Component} from 'react';**  **const jwt = require("jsonwebtoken"); //**  **class EnterWeightMain extends Component{** |

1. Before we do anything else, add a new function into enterweightmain.js, actually this function, **componentDidMount()** is part of the lifecycle of a component in ReactJS

|  |
| --- |
|  |

In **componentDidMount**() the first two lines are exactly the same as in **auth.js** from our NodeJS application. We first read the token then decode it. Once we decode we get the payload. Remember the payload contains lots of information we can use in any part of the app.

1. Now we can use the payload details to assign to the state object, here is the entire function

|  |
| --- |
| **componentDidMount() {**  **const rawToken = localStorage.getItem('token');**  **const decToken = jwt.verify(rawToken, 'mysecret');**  **if(decToken){**  **this.setState({empName: decToken.empName});**  **this.setState({loginStatus: "Logged in"});**  **}**  **}** |

There is a possibility that if the token is not present on the browser, this code will throw an error. A better implementation is shown below:

|  |
| --- |
| **componentDidMount() {**  **const self = this;**  **const rawToken = localStorage.getItem('token');**  **jwt.verify(rawToken, 'mysecret', function(err, decoded) {**  **if (err) {**  **self.setState({feedback: 'Failed to authenticate token.'});**  **} else {**  **const decToken = jwt.verify(rawToken, 'mysecret');**  **}**  **});**  **}** |

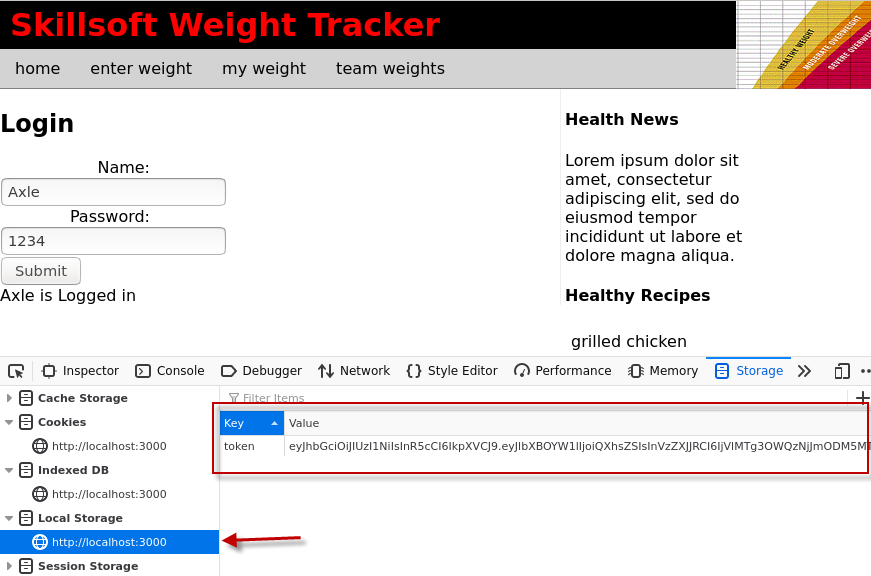
1. If you did duplicate the **login** component, then we need to change the *fetch* url in **handleSubmit()**. Also this is a PUT request not a POST. We created an **addnewweight** endpoint on Day02 Part07.

|  |
| --- |
| **handleSubmit(event) {**  **const self = this;**  **event.preventDefault();**  **fetch('http://localhost:8000/addnewweight', {**  **method: 'PUT',**  **headers: {**  **'Accept': 'application/json',**  **'Content-Type': 'application/json'**  **},**  **body: JSON.stringify({**  **empName: this.state.empName,**  **empWeight: this.state.empWeight**  Remove the other two lines from this area  **})**  **}).then(function(response){**  **return response.json()**  **}).then(function(data){**  **self.setState({feedback: data.message});**  **}).catch(function(err){**  **console.log(err);**  **})**  **}**  **//** |

The feedback name value pair is what our NodeJS server will respond with, once we have a successful PUT into our database. At this point, you may choose to send a different ‘view’ to the user so that they don’t enter the same information twice.

1. Using our jwt, we can display details like *empName, loginStatus* and *feedback* in the HTML. **empName** was handled in the previous section.

|  |
| --- |
| **</label>**  **</div>**  **<div>**  **<input type="submit" value="Submit" />**  **</div>**  **<div>**  **<span>{this.state.loginStatus}</span>**  **</div>**  **<div>**  **<span>{this.state.feedback}</span>**  **</div>**  **</form>**  **</main>** |

1. Now login once more using /login, then attempt to go to /enterweight, now we should see the user logged in. Also in the browser tools, if you look at Local Storage, you should see our token stored there.  
   
2. You can try to enter a weight, I entered 81 and this is a screenshot of the MongoDB feedback using find(). Remember to use /enterweight  
   

# part 07 – Adding Links to Menu Items

We will implement simple routing using just the header.js file (in wrapper)

1. In header.js import the Link component from react-router-dom (Wrapper folder)

|  |
| --- |
| **import React from "react";**  **import { Link } from 'react-router-dom';**  **import logo from '../../chart.gif';**  **function Header(){** |

1. Now all we need to do is use the **Link to** syntax to point our various menu items to the correct ‘view’

|  |
| --- |
| **<nav>**  **<ul>**  **<li><Link to="/home">home</Link></li>**  **<li><Link to="/enterweight">enter weight</Link></li>**  **<li><Link to="/">my weight</Link></li>**  **<li><Link to="/teamweights">team weights</Link></li>**  **</ul>**  **</nav>** |

I have no implemented **my weight**, so there is no link for that menu item yet.

1. At this point, the links should work. Since we are using just the header.js for navigation, there is nothing else to do. However we still have login.
2. Just copy the last <li> line and replace the content accordingly

|  |
| --- |
| **<ul>**  **<li><Link to="/home">home</Link></li>**  **<li><Link to="/enterweight">enter weight</Link></li>**  **<li><Link to="/">my weight</Link></li>**  **<li><Link to="/teamweights">team weights</Link></li>**  **<li><Link to="/login">login</Link></li>**  **</ul>** |

1. Add a style to push the login menu item far to the right

|  |
| --- |
| **nav li:last-of-type{**  **float: right;**  **}** |

1. Your view should look like this

|  |
| --- |
|  |

1. Click the login button, the browser should show the login view from Part04 above.

# part 08 – Fixing team weights

Back on Day03, parts 7 and 8 we configured **teamweights**, but now that we have implemented security, this view will not render properly. In fact, if you try to go to /teamweights you would see the message ‘Not authorized’. We would need to pass the token in the **fetch()** request in order to get proper results.

1. Using Atom, go to weightsmain.js, which is inside of the teamweights folder and scroll down to the **ComponentDidMount()** method. Add a second parameter to hold our headers.

|  |
| --- |
| **//**  **componentDidMount(){**  **fetch("http://localhost:8000/getweights",{})**  **.then(response => response.json())**  **.then(response => {** |

1. Although the GET method is implied when we make a request, we can be explicit so add a new name value pair for our request method:

|  |
| --- |
| **icomponentDidMount(){**  **fetch("http://localhost:8000/getweights",{**  **method:'GET'**  **})**  **.then(response => response.json())** |

1. Do the same for headers, but headers is itsellf an object

|  |
| --- |
| **componentDidMount(){**  **fetch("http://localhost:8000/getweights",{**  **method:'GET',**  **headers:new Headers({})**  **})**  **.then(response => response.json())** |

1. Finally we can finish the Headers object with a couple more name-value pairs, one for authorization where we send the token and the second for the type of content we are workign with, so json in this case.

|  |
| --- |
| **componentDidMount(){**  **fetch("http://localhost:8000/getweights",{**  **method:'GET',**  **headers:new Headers({**  **'Authorization': 'Bearer '+localStorage.getItem('token'),**  **'Content-Type': 'application/x-www-form-urlencoded'**  **})**  **})**  **.then(response => response.json())** |

1. Login and try the route /teamweights, it shoud now show the weights for Axle, the only user we have so far

|  |
| --- |
|  |

Here is the entire **componentDidMount()** method

|  |
| --- |
| **componentDidMount(){**  **fetch("http://localhost:8000/getweights", {**  **method:'GET',**  **headers:new Headers({**  **'Authorization': 'Bearer '+localStorage.getItem('token'),**  **'Content-Type': 'application/x-www-form-urlencoded'**  **})**  **})**  **.then(response => response.json())**  **.then(response => {**  **this.setState({**  **allWeights:response**  **})**  **})**  **} .then(response => response.json())** |

# part 09 – Fixing team weights

On Day01, Part06 we introduced an API for creating a new employee document. We now need to add a document using our ReactJS application, to facilitate adding a new employee. We would need a form with two fields and then POST the values to our API from Day01.

1. Copy/duplicate the entire enterweight folder and rename the folder addnewemployee
2. Rename the files to employeemain.js, employeecontainer.js and addemployee.js
3. Starting with addemployee.js rename the fields and import the corresponding components:

|  |
| --- |
| **import React from "react";**  **import EmployeeContainer from "./employeecontainer";**  **import Footer from "./../wrappers/footer";**  **import Header from "./../wrappers/header";**  **function AddEmployee(){**  **return (**  **<div>**  **<Header />**  **<EmployeeContainer />**  **<Footer />**  **</div>**  **)**  **}**  **export default AddEmployee** |

1. Then employeecontainer.js

|  |
| --- |
| **import React from 'react';**  **import EmployeeMain from "./employeemain";**  **import Aside from "./../wrappers/aside";**  **function EmployeeContainer (){**  **return(**  **<div>**  **<EmployeeMain />**  **<Aside />**  **</div>**  **)**  **}**  **export default EmployeeContainer ;** |

1. Then in employeemain.js.

|  |
| --- |
| **class EmployeeMain extends Component{**  **constructor(props) {**  **super(props); }**  **…….**  **export default EmployeeMain;** |

1. Also change the prompt between the <h2> tags:

|  |
| --- |
| **class EmployeeMain extends Component{**  **constructor(props) {**  **super(props); }**  **…….**  **export default EmployeeMain;** |

Leave all the other code from **enterweight** for now

1. Add a route to this view in App.js

|  |
| --- |
| **import React from 'react';**  **import './styles.css';**  **import Home from './components/home/home';**  **import AddEmployee from './components/addemployee/addemployee';**  **import Teamweights from './components/teamweights/teamweights';**  ……..  **<Route exact path="/" component={Home} />**  **<Route path="/home" component={Home} />**  **<Route path="/addemployee" component={AddEmployee} />**  **<Route path="/teamweights" component={Teamweights} />** |

1. Remove the first pair of **<div>** tags and duplicate the second pair

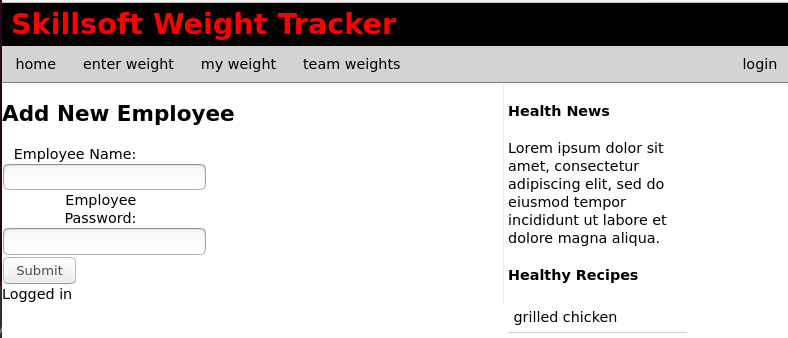
|  |
| --- |
| **<form onSubmit={this.handleSubmit}>**  **<div>**  **<label>**  **Weight today:**  **<input type="text" name="empWeight" onChange={this.handleFieldChange}/>**  **</label>**  **</div>**  **<div>**  **<label>**  **Weight today:**  **<input type="text" name="empWeight" onChange={this.handleFieldChange}/>**  **</label>**  **</div>**  **<div>** |

1. Change the labels and fields to reflect **empName** and **empPass**

|  |
| --- |
| **<form onSubmit={this.handleSubmit}>**  **<div>**  **<label>**  **Employee Name:**  **<input type="text" name="empName" onChange={this.handleFieldChange}/>**  **</label>**  **</div>**  **<div>**  **<label>**  **Employee Password:**  **<input type="text" name="empPass" onChange={this.handleFieldChange}/>**  **</label>**  **</div>**  **<div>**  **<input type="submit" value="Submit" />** |

1. If we are adding a new document, we do not need any JWT interaction, so we can remove the content from componentDidMount(), unless of course you make this part of your *admin* functionality and requires sign-in.

|  |
| --- |
| **loginStatus:"",**  **feedback:""**  **}**  **}**  **componentDidMount() {**  **}**  **handleSubmit(event) {**  **const self = this;** |

Test the component, it should look like this  


# part 10 – Adding a new Employee (API)

We would need to change our endpoint and perform an automatic login, when we accept a new employee into our system.

1. First in employeemain.js comment out or delete the **componentDidMount()** method.
2. Change the HTTP verb to POST and the endpoint to addnewemployee

|  |
| --- |
| **handleSubmit(event) {**  **const self = this;**  **event.preventDefault();**  **fetch('http://localhost:8000/addnewemployee', {**  **method: 'POST',**  **headers: {**  **'Accept': 'application/json',**  **'Content-Type': 'application/json'** |

1. In the body name value section, change **empWeight** to **empPass**

|  |
| --- |
| **'Content-Type': 'application/json'**  **},**  **body: JSON.stringify({**  **empName: this.state.empName,**  **empPass: this.state.empPass**  **})**  **}).then(function(response){** |

1. Prepare the **state** object for reflecting a successful post

|  |
| --- |
| **this.handleFieldChange = this.handleFieldChange.bind(this);**  **this.state = {**  **empName:"",**  **empWeight:"",**  **postStatus:"",**  **token:””**  **} }).then(function(response){** |

Rememember to cater for the token, otherwise the new employee will not be logged in automatically.

1. In the second **then()** method, change the status to reflect what the API provides

|  |
| --- |
| **}).then(function(response){**  **return response.json();**  **}).then(function(data){**  **self.setState({postStatus: data.message});  self.setState({token: data.token});  localStorage.setItem('token', self.state.token)**  **}).catch(function(err){**  **console.log(err);** |

1. Show a successful post by displaying the value of postStatus in a pair of **<div>** tags like we did for other forms. Remove the login pair of **<div>** tags or comment then out for now.

|  |
| --- |
| **<div>**  **<input type="submit" value="Submit" />**  **</div>**  **<div>**  **<span>{this.state.postStatus}</span>**  **</div>**  **</form>** |

1. Test by adding a new employee

|  |
| --- |
|  |

John was added correctly, but he has no weights data as yet.

# part 11 – Adding a new Employee (Login Status)

Usually, after the new employee is created, we would probably want that user to be automatically logged in. In this way they can enter a weight right away. Again, several approaches are possible, but we will use code that we already have.

1. Back in the NodeJS application, if you look at the **loginuser()** method of controller.js, the **jwt.sign()** method is what generates a token. We can apply that piece of code to our **addnewemployee()** method

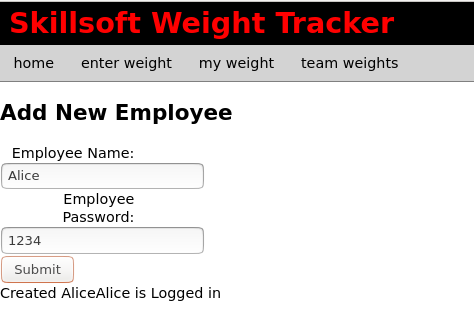
|  |
| --- |
| **Emp.save({}, function(err){**  **if (err)**  **response.end(err);**  **//no error so get token**  **jwt.sign({**  **empName:results[0].empName,**  **userID:results[0].\_id**  **},**  **"mysecret",**  **{expiresIn : "1h"},**  **function(err, token){**  **if(err) throw err;**  **response.json({token:token});**  **}**  **)**  **response.json({message:`Created ${Emp.empName}`});** |

1. This code will not work, because we did not get anything in *results*, we would need to use *Emp*, the object, so change a couple of lines

|  |
| --- |
| **jwt.sign({**  **empName:Emp.empName,**  **userID:Emp.\_id**  **},**  **"mysecret",**  **{expiresIn : "1h"},** |

1. Now in the **response.json()** method, just send back the token, instead of the message or in addition to the message. Also comment out the **response.end()** line if you still have it.

|  |
| --- |
| **jwt.sign({**  **empName:Emp.empName,**  **userID:Emp.\_id**  **},**  **"mysecret",**  **{expiresIn : "1h"},**  **function(err, token){**  **if(err) throw err;**  **response.json(**  **{message:`Created ${Emp.empName}`, token:token}**  **);**  **}**  **)** |

1. Test it out  
   

# part 12 – Adding a New Search API (Optional)

Head back over to your Node application and we will copy one of the previous functions and make it a search function

1. Copy the **getweights()** function and rename it to **getbyname().**

|  |
| --- |
| **exports.getbyname = function(request, response){**  **Employee.find({}, function(err, results){**  **if (err)**  **response.end(err);**  **response.json(results);**  **});**  **};** |

1. Change the function to the following. First we create a new variable and assign it whatever value the user sends via the request body. Then we pass that value on to the **find()** function of Mongodb. The rest is just like in the **getweights()** function. The only difference is that we are seeking one document. Also note that we are handling the name value passed via the *parameter* of the request object, so in the URL.

|  |
| --- |
| **exports.getbyname = function(request, response){**  **let empToFind = request.params.employeeName;**  **Employee.find({empName:empToFind}, function(err, results){**  **if (err)**  **response.end(err);**  **response.json(results);**  **});**  **};** |

1. We also need to add a route to the route.js file. Note, do not use the auth middleware, you can add that in later.

|  |
| --- |
| **pp.route('/getweights').get(auth, controller.getweights);**  **app.route('/getbyname/:employeeName').get(controller.getbyname);**  **app.route('/addnewemployee').post(controller.addnewemployee);**  **app.route('/addnewweight').put(controller.addnewweight);**  **app.route('/loginuser').post(controller.loginuser)** |

1. Test this functionality using Postman or the Browser

|  |
| --- |
|  |

Notice that we got the entire document belonging to Axle. Now its time to test the API using the ReactJS app.

# part 13 – Implement Search API in ReactJS PartA (Optional)

In react we can combine the login in getweights and any of the forms to get the result for one employee.

1. Create a new component to correspond with the menu item of ‘my weight’. You can do this by copying any of the other components and renaming the appropriate parts. Here is my layout. I choose to duplicate the home component as it is fairly simple. Also the image shows the myweightscontainer.js file completed.

|  |
| --- |
|  |

1. Starting with the myweightscontainer.js file, change the naming accordingly.

|  |
| --- |
| **import React from "react";**  **import MyWeightsMain from "./myweightsmain";**  **import Aside from "./../wrapper/aside";**  **function MyWeightsContainer(){**  **return (**  **<div>**  **<MyWeightsMain/>**  **<Aside />**  **</div>**  **)**  **}**  **//**  **export default MyWeightsContainer** |

1. We also need myweights.js file completed.

|  |
| --- |
| **import React from "react";**  **import MyWeightsContainer from "./myweightscontainer";**  **import Footer from "./../wrapper/footer";**  **import Header from "./../wrapper/header";**  **//**  **function MyWeights(){**  **return (**  **<div>**  **<Header />**  **<MyWeightsContainer />**  **<Footer />**  **</div>**  **)**  **}**  **//**  **export default MyWeights** |

1. The most important file is the myweightsmain.js file, here I have just done some minor changes.

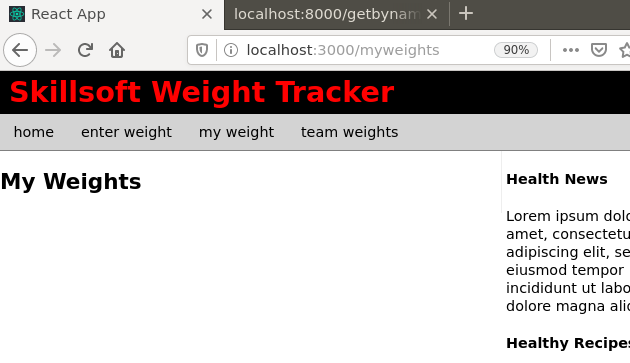
|  |
| --- |
| **import React from "react";**  **//**  **function MyWeightsMain(){**  **return (**  **<main>**  **<h2>My Weights</h2>**  **</main>**  **)**  **};**  **//**  **export default MyWeightsMain** |

1. Add a route and a menu item

|  |
| --- |
| **import Login from './components/login/login';**  **import EnterWeight from './components/enterweight/enterweight';**  **import MyWeights from './components/myweights/myweights';**  **import {BrowserRouter, Route} from 'react-router-dom';**  …  **<Route path="/login" component={Login}/>**  **<Route path="/enterweight" component={EnterWeight} />**  **<Route path="/myweights" component={MyWeights} />**  **</BrowserRouter>** |

1. Now we can complete the menu <ul> on header.js

|  |
| --- |
| **import Login from './components/login/login';**  **import EnterWeight from './components/enterweight/enterweight';**  **import MyWeights from './components/myweights/myweights';**  **import {BrowserRouter, Route} from 'react-router-dom';**  …  **<Route path="/login" component={Login}/>**  **<Route path="/enterweight" component={EnterWeight} />**  **<Route path="/myweights" component={MyWeights} />**  **</BrowserRouter>** |

1. If you test the app now you should see something like this  
   

# part 14 – Implement Search API in ReactJS PartB (Optional)

We will continue to develop myweightsmain.js as it will contain all the logic for searching via a form and displaying details about what was found. This class is being developed a little differently. Instead of having one large div to return, we split up pieces of HTML. We will have two mini components, one for displaying the form and one for dealing with data from our API hit.

1. Start by turning the **MyWeightsMain** function into a class, give it a constructor and reference props. Also setup a state object.

|  |
| --- |
| **import React, {Component} from 'react';**  **//**  **class MyWeightsMain extends Component{**  **constructor(props) {**  **super(props);**  **this.state = {**  **empName: '',**  **allWeights: []**  **}**  **}**  **render(){**  **return (**  **<main>**  **<h2>My Weights</h2>**  **</main>**  **)**  **}**  **};**  **//**  **export default MyWeightsMain** |

1. In the **render()** method, instead of returning just HTML, we will return a separate *component* that renders HTML. Lets first define that functional component which is actually a const called **Search**. **Search** is the first mini component that will render the HTML form. Search is inside of myweightsmain.js file.

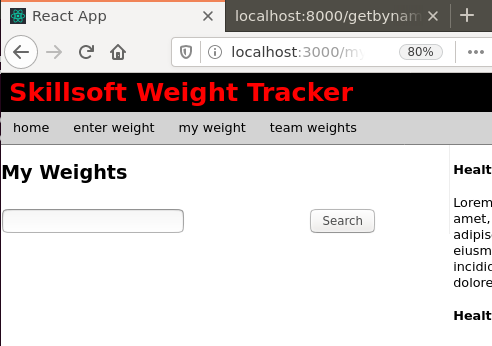
|  |
| --- |
| **import React, {Component} from 'react';**  **//**  **const Search = (props) => {**  **return <div>**  **<input**  **type="text"**  **value={""}**  **onChange={""}**  **/>**  **<button onClick={""}>Search</button>**  **</div>;**  **}**  **//**  **class MyWeightsMain extends Compone** |

In this **const**, we reference **props**. We return a pair of parent **<div>** tags that enclose an **input** of type text. This input box will have a *value* and an *onChange* event. It will also have a **button** which itself has an *onClick* event which we will hookup soon.

1. Lets now reference **Search** from the **render()** method of **MyWeightsMain**

|  |
| --- |
| **render(){**  **return (**  **<main>**  **<h2>My Weights</h2>**  **<Search />**  **</main>**  **)**  **}**  **};** |

The view should now look like this:



1. In our **Search** component, we define use the value in **empName**, an **onChange** event and a search property that will reference the **onSearch** event of the **Search** component.

|  |
| --- |
| **import React, {Component} from 'react';**  **//**  **const Search = (props) => {**  **const {**  **empName,**  **onChange,**  **search**  **} = props;**  **//**  **return <div>** |

1. Now I can populate the **input** box with the proper values and functions.

|  |
| --- |
| **} = props;**  **//**  **return <div>**  **<input**  **type="text"**  **value={ empName }**  **onChange={onChange}**  **/>**  **<button onClick={search}>Search</button>**  **</div>;** |

1. In the **render()** method of **MyWeightsMain**, we can now complete the **Search** component with its values and functions

|  |
| --- |
| **render(){**  **return (**  **<main>**  **<h2>My Weights</h2>**  **<Search**  **searchQuery={** **empName }**  **onChange={this.onSearchQueryChange}**  **search={this.onSearch}**  **/>**  **</main>**  **)**  **}** |

1. We have to define the two events in the constructor of our class. We did this for all our forms so far.

|  |
| --- |
| **this.state = {**  **empName: '',**  **allWeights: []**  **}**  **this.onSearchQueryChange = this.onSearchQueryChange.bind(this);**  **this.onSearch = this.onSearch.bind(this);**  **}**  **render(){**  **return** |

1. We need to methods to perform the search work, we will star the onSearch and onSearchQueryChange methods. Place these just above the render method.

|  |
| --- |
| **onSearchQueryChange(e) {**  **}**  **onSearch() {**  **}** |

1. In the onSearchQueryChange method, set the value of **empName** from our form field via the **onSearchQueryChange** function

|  |
| --- |
| **this.onSearchQueryChange = this.onSearchQueryChange.bind(this);**  **this.onSearch = this.onSearch.bind(this);**  **}**  **onSearchQueryChange(e) {**  **this.setState({empName: e.target.value});**  **}**  **render(){**  **return (** |

1. We also need to define the **onSearch** method. This is the method that will access our *getbyname* endpoint and return all the data. We define **onSearch** in the class itself. The **onSearch** method here is almost an exact copy of the one used for **teamweights**

|  |
| --- |
| **onSearch() {**  **fetch("http://localhost:8000/getbyname/" + this.state.empName,**  **{**  **method:'GET',**  **headers: new Headers({**  **'Authorization': 'Bearer '+localStorage.getItem('token'),**  **'Content-Type': 'application/x-www-form-urlencoded'**  **}),**  **}**  **)**  **.then(response => response.json())**  **.then(response => {**  **this.setState({**  **allWeights:response**  **})**  **})**  **}** |

1. At this point you are probably getting an error about **empName** not being defined. Actually we have to destructure **empName** and **allWeights** within our **render()** method, so that method has access to both of these variables

|  |
| --- |
| **render(){**  **const {empName, allWeights} = this.state;**  **return (**  **<main>** |

1. If you test the **myweight** feature now, it will simply return an empty object. Just like we created a component using **Search** as a **const**, we can do the same to handle the response from our API hit. Create a new component with **const**, call it **AllWeights** with uppercase **A** and **W** and point it to a function that takes **allWeights** (from our State object) as an argument

|  |
| --- |
| **<button onClick={search}>Search</button>**  **</div>;**  **}**  **const AllWeights = ({allWeights}) => {**    **}**  **class MyWeightsMain extends Component{** |

I am doing this below the **Search** component but just above the start of the class

1. What we return from this component is almost exactly what we returned from the **render()** function in weightsmain.js (teamweights)

|  |
| --- |
| **const AllWeights = ({allWeights}) => {**  **return allWeights.map((emp, i) =>**  **(**  **<div key={i}>**  **{emp.empName} {**  **emp.employeeWeights.map(function(weight, j){**  **return <div key={j}>**  **Date: {new Date(weight.weighedDate).toLocaleDateString()}**  **{' '}**  **Weight: {weight.empWeight}**  **</div>**  **})**  **}**  **</div>**  **)**  **);**  **}** |

1. Now all we have to do in our **render()** method is invoke this **AllWeights** componenet below the **<Search />** component. Remember we had de-structured the **allWeights** variable so that we can access its contents from our **State** object. We did this on step 10 above.

|  |
| --- |
| **render(){**  **const {empName, allWeights} = this.state;**  **return (**  **<main>**  **<h2>My Weights</h2>**  **<Search**  **searchQuery={empName}**  **onChange={this.onSearchQueryChange}**  **search={this.onSearch}**  **/>**  **<AllWeights allWeights={allWeights} />**  **</main>**  **)**  **}** |

1. Here is the result of searching for Axle

