Table of Contents

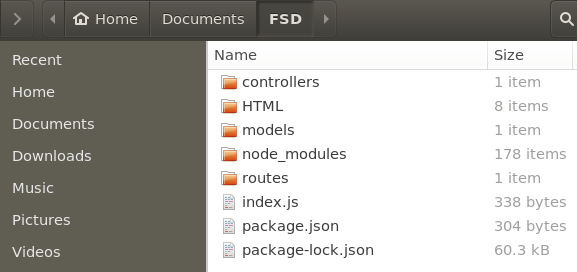
[**part 1 – Folder Setup** 2](#_Toc19262643)

[**part 2 – Configuring Team Weights** 4](#_Toc19262644)

[**part 3 – adding a new record (jQuery)** 5](#_Toc19262645)

Day03 Deployment

## part 1 – Folder Setup

1. Insert the HTML folder from Day02 into your API folder so that HTML is at the root of API.  
   
2. Run the nodemon command to spin the app
3. First task is to serve the **HTML** pages we built on **Day02**. Now that we have a **controller** file and we are using Express router, everything that goes to the client will pass through the appropriate controller function. What we want is to serve our HTML page when users land on our root which at the moment is <http://localhost/8000>. Right now we are serving up plain text.
4. We would need to change line 3 of controller.js (getdefault) to serv the index.html file instead, which now is inside of the HTML folder.

|  |
| --- |
| **const Weight = require('../models/employees);**  **exports.getdefault=function(req, res){**  **res.send('../HTML/index.html');**  **};** |

1. Of course this will not work because whatever is between the single quotes will simply print out on the browser screen. Node has a different function to handle html files, it is called **sendFile()** and it is attached to the response object.

|  |
| --- |
| **const Weight = require('../models/employees'); exports.getdefault=function(req, res){**  **res.sendFile(‘../HTML/index.html’);**  **};** |

1. This did not work but it provided some clues, something about a path. In order to serve static pages, we need the **path** package, so in the controller.js file, declare a variable and point it to the **path** package.

|  |
| --- |
| **const Weight = require('../models/employees'); const path = require("path");**  **exports.getdefault=function(req, res){** |

1. The path object has a method called **join()** which we can use to obtain the current path of the application. If we then concatenate the root path with the path where our **HTML** files live, we can finally obtain a true absolute path to our files

|  |
| --- |
| **const path = require("path");**  **exports.getdefault=function(req, res){**  **res.sendFile(path.join(\_\_dirname + '/../HTML/index.html'));**  **};** |

1. Although the html file is served, it appears to not know that CSS and JS exists, we need to let Express know that these files exist and that it should use them. To serve static files such as images, CSS files, and JavaScript files, Express uses a static() method, which is passed in as middleware, so in index.js:

|  |
| --- |
| **app.use('/', router);**  **app.use(express.static('HTML'))**  **//**  **app.listen(port, function(){** |

1. So now at our root url, we get HTML and not just a sentence. Try the team weights link and you will see that it works just like before.

## part 2 – Configuring Team Weights

1. We did not have to do much work with teamweights.html, but we can improve the display a bit. Open that html file and go to where the **h2** element is, there should be a pair of **div** tags with an id of **records**, if not add it now.

|  |
| --- |
| **<main>**  **<h2>Showing records for team</h2>**  **<div id="records"></div>**  **<button id="getData">Get Records</button>**  **</main>**  **<aside>** |

1. In the JS file, look for the displayData() function and where we had <p> tags before change them to <div> tags

|  |
| --- |
| **function displayData(arr) {**  **let outHTML = "";**  **for(let i=0; i < arr.length; i++){**  **outHTML+="<div>"+arr[i].empName + " weighed " + arr[i].empWeight + " Kgs</div>";**  **}**  **document.getElementById("records").innerHTML = outHTML;**  **}** |

1. Now we can target that id in the css, so create a new style for our display of records:

|  |
| --- |
| **#records {**  **margin-left:34px;**  **width:80%;**  **}** |

Refresh the teamweights.html file and adjust the CSS to your liking.

1. We may also want to change the background color of alternating rows just for easier reading, again you can play with the background colors of this code:

|  |
| --- |
| **#records div:nth-child(odd) {**  **background: lightgray;**  **display: block;**  **margin:5px 0px;**  **}** |

1. (Optional) We can get the same effect if we employ the more modern map() of JavaScript:

|  |
| --- |
| **function displayData(arr) {**  **arr.map(arr=>{**  **document.getElementById("records").innerHTML+="<div>"+arr.empName+"</div>";**  **})**  **}** |

With the map() method, we write less code. I only included the name here but you can easily expand this to include the weight and the date.

## part 3 – Installing and Configuring JWT

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

**npm install jsonwebtoken**

You can restart the application using **nodemon**

1. Go into the **models** folder and copy the employees.js file paste and rename to users.js
2. Change the users.js file contents to reflect a user instead of weights:

|  |
| --- |
| **const wSchema = new mongoose.Schema({**  **empName: String,**  **empPass: String,**  **created: {type: Date, default: Date.now }**  **},{**  **collection:'EmployeeUsers'**  **});**  **module.exports = mongoose.model('Users', wSchema);** |

1. In controllers.js file import the **EmployeeUsers** collection, the same way we imported the Weights collection

|  |
| --- |
| **const Weight = require('../models/employees');**  **const User = require('../models/users');**  **const path = require("path");** |

1. In controllers.js file, copy the function **addnewdoc** and change its name to **addnewuser**. Also change the contents of **addnewuser** to reflect adding users instead of employee weights

|  |
| --- |
| **exports.addnewuser=function(req, res){**  **let empName = req.body.empName;**  **let empPass = req.body.empPass;**  **const user = new User();**  **user.empName = empName;**  **user.empPass = empPass;**  **user.save({}, function(err){**  **if(err)**  **res.end(err);**  **res.end(`Created ${empName}`)**  **});**  **};** |

1. Also import the **jsonwebtoken** package

|  |
| --- |
| **const jwt = require('jsonwebtoken');**  **const Weight = require('../models/employees');**  **const User = require('../models/users');** |

1. In controllers.js file, copy the **addnewuser** function and rename it to **loginuser**. This function will handle logging in of users. There is no need to logout a user with a JWT solution, the token simply expires. Also remove everything except the first two lines.

|  |
| --- |
| **exports.loginuser = function(req,res){**  **let empName = req.body.empName;**  **let empPass = req.body.empPass;**  **};** |

1. Now implement the **find()** function to find the user seeking access (or a token in this case)

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

1. If we find the user, the user’s details should be in results. If we do not find the user we send back to the client that the login failed. Notice I am just checking the passwords here

|  |
| --- |
| **exports.loginuser = function(req,res){**  **let empName = req.body.empName;**  **let empPass = req.body.empPass;**  **User.find({empName:empName}, function(err, results){**  **if (err)**  **res.end(err);**  **if(results[0].empPass == empPass){**  **//user exists so now use jsonwebtoken**  **} else {**  **res.end("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

|  |
| --- |
| **User.find({empName:empName}, function(err, results){**  **if (err)**  **res.end(err);**  **if(results[0].empPass == empPass){**  **jwt.sign({**  **empName:results[0].empName,**  **userID:results[0].\_id**  **},**  **"mysecret",**  **{expiresIn : "1h"},**  **function(err, token){**  **if(err) throw err;**  **res.end(token);**  **})**  **} else {**  **res.end("Login failed");**  **}**  **});** |

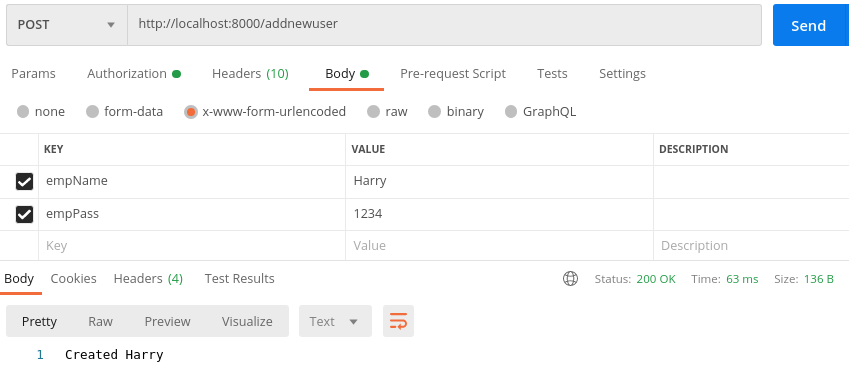
The sign() method takes several parameters. Once find() from mongo returns, it will contain the name and password, so we extract that using the array syntax and passing that as a payload to the sign() method of jwt. We also pass a some text which will act as our key, then an expiry date and finally a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

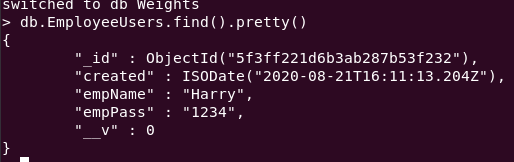
Here is the entire loginuser() function:

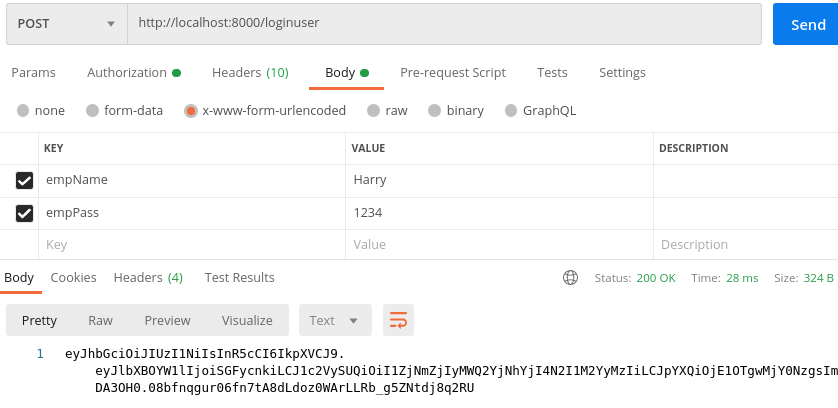
|  |
| --- |
| **exports.loginuser=function(req, res){**  **let empName = req.body.empName;**  **let empPass = req.body.empPass;**  **User.find({empName:empName}, function(err, results){**  **if (err)**  **res.end(err);**  **if(results[0].empPass == empPass){**  **jwt.sign({**  **empName:results[0].empName,**  **userID:results[0].\_id**  **},**  **"mysecret",**  **{expiresIn : "1h"},**  **function(err, token){**  **if(err) throw err;**  **res.end(token);**  **})**  **} else {**  **res.end("Login failed");**  **}**  **});**  **};** |

1. In routes.js file, add two routes to handle user signup and login, the controller functions already exist. Make sure that they are POST routes:

|  |
| --- |
| **router.put('/updatedoc', controller.updatedoc);**  **//**  **router.post('/addnewuser', controller.addnewuser);**  **//**  **router.post('/loginuser', controller.loginuser);** |

1. Test adding a new user using the REST client  
   
2. You may also verify by logging into the MongoDB CLI and checking for the EmployeeUsers collection. We are using MongoDB directly as we do not yet have an API that could produce this sort of information like /getdocs:



1. 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 method:  
     
   

## part 4 – 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. 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 file of a request. 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.

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. Since this is middleware, we have access to the response and request objects, we could pass back to controller via the request object the token we just received, although it is not necessary in this case. Also 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');**  **req.userInfo = decToken;**  **next();**  **}**  **}** |

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

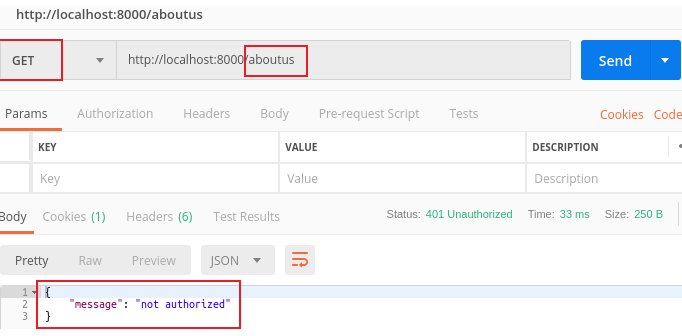
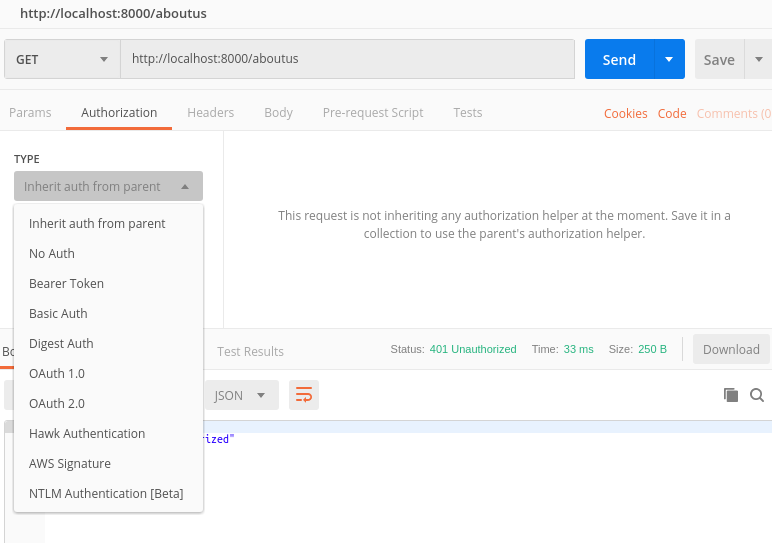
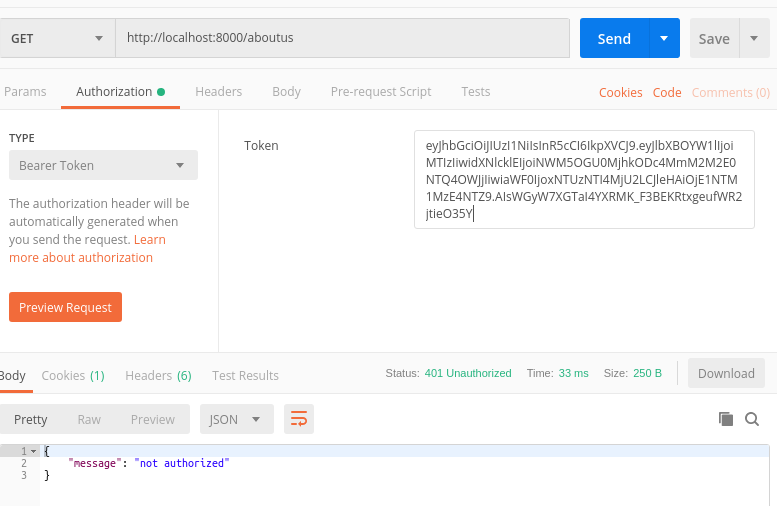
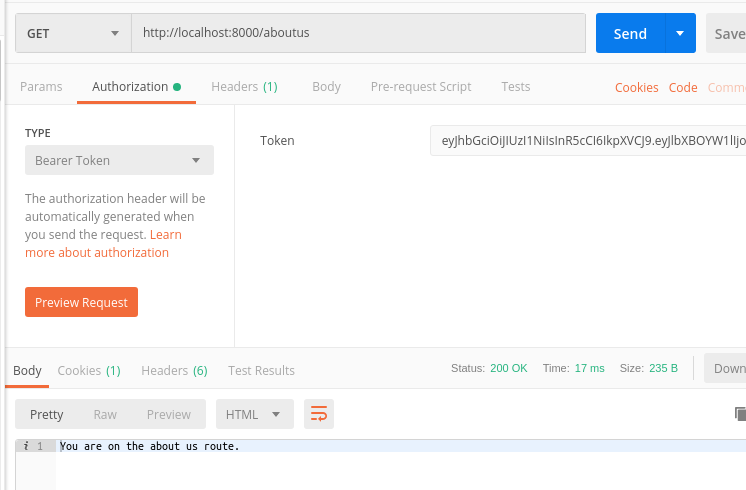
|  |
| --- |
| **const jwt = require("jsonwebtoken");**  **module.exports = function (req, res, next) {**  **try{**  **const rawToken = req.headers.authorization.split(" ")[1];**  **const decToken = jwt.verify(rawToken, 'mysecret');**  **req.userInfo = decToken;**  **next();**  **}catch(error){**  **return 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 controller = require('../controllers/controller');**  **let authUser = require('../controllers/auth');**  **module.exports = function(router){**  **//**  **router.get('/', controller.getdefault);** |

1. We will experiment with the aboutus route, in terms of protecting this route. Simply insert the authUser variable before the controller part

|  |
| --- |
| **router.get('/', controller.getdefault);**  **//**  **router.post('/addweight', controller.addweight);**  **//**  **router.get('/aboutus', authUser, controller.aboutus);**  **//**  **router.get('/getdocs', controller.getdocs);** |

1. Now we can test first without the token:  
   
2. Now lets make the same request by passing in the token we generated on a previous tab  
     
   
3. Choose Bearer Token and you will get a small box to enter the token from a previous tab.   
     
   11. Now you can hit the send button  
   
4. You may try to manually change the token, for example remove the first “e” and hit send, the request will be denied

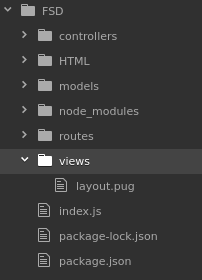
## part 5 – Install pug and start to build a Template

1. Run the following command to install Pug (remember to stop the application first)

**npm install pug --save**

1. Let the Express app know that we will be using Pug by using the set() method. This is done in the index.js file

|  |
| --- |
| **const port = 8000;**  **const app = express();**  **app.use(bodyParser.urlencoded({extended:false}));**  **app.set('view engine', 'pug');**  **const router = express.Router();**  **const routes = require('./routes/routes');** |



1. Express expects that a **views** folder exists which will store all the templates, so create that folder now inside of the root folder, and then inside of that views folder create a text file named **layout** with a file extension of .**pug** (so the name of the file is layout.pug)
2. Start building the layout inside of layout.pug using similar syntax to HTML

|  |
| --- |
| **html**  **head**  **title**  **body**  **header**  **h1**  **nav**  **ul**  **li**  **a(href='index.html') home**  **div#container**  **block content**  **footer** |

1. Create a new route in routes.js to point to a pug test page

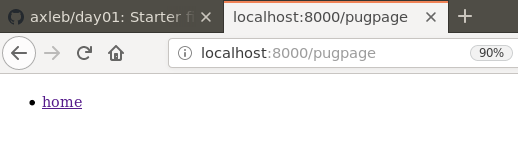
|  |
| --- |
| **router.route('/loginuser').post(controller.loginuser);**  **//**  **router.route('/pugpage').get(controller.pugpage);**  **//**  **};** |

1. Then in the controllers.js file, add a method to handle this test route, just copy the **aboutus** function and comment the code that’s already there:

|  |
| --- |
| **//**  **exports.pugpage=function(req, res){**  **//res.send('You are on the about us route.');**  **};** |

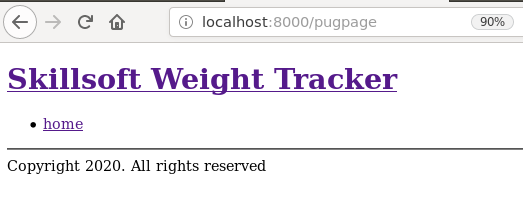
1. Now simply call the *render* method instead of the *send* method and pass in the name of the pug file (without the .pug extension)

|  |
| --- |
| **exports.pughome=function(req, res){**  **//res.send('You are on the about us route.');**  **res.render(**  **'layout'**  **)**  **};** |

You should see somehting like this  


1. Add some content so that we can test this layout.pug file

|  |
| --- |
| **doctype html**  **html**  **head**  **title Skillsoft Weight Tracker**  **body**  **header**  **h1**  **a(href='index.html') Skillsoft Weight Tracker**  **nav**  **ul**  **li**  **a(href='index.html') home**    **div#container**  **block content**  **footer**  **hr**  **| Copyright 2020. All rights reserved** |



## part 6 – completing the pug Layout template

1. Complete the layout template to include the entire navigation, css and script tags

|  |
| --- |
| **doctype html**  **html**  **head**  **title Skillsoft Weight Tracker**  **link(rel='stylesheet', type='text/css', href='styles/styles.css')**  **body**  **header**  **h1**  **a(href='index.html') Skillsoft Weight Tracker**  **nav**  **ul**  **li**  **a(href='index.html') home**  **li**  **a(href='enterweight.html') enter weight**  **li**  **a(href='myweights.html') my weight**  **li**  **a(href='teamweights.html') team weights**  **div#container**  **block content**  **footer**  **hr**  **| Copyright 2020. All rights reserved**  **script(src='scripts/scripts.js')** |

Test the layout route again.

1. In order to demonstrate how this layout will be helpful, we would now simulate the creation of a new web page for our website. We will create a new .pug file in the views folder and call it pughome.pug for now. When we create new pages from now on, we simply include the wrapper or layout template by extending it.

|  |
| --- |
| **extends layout** |

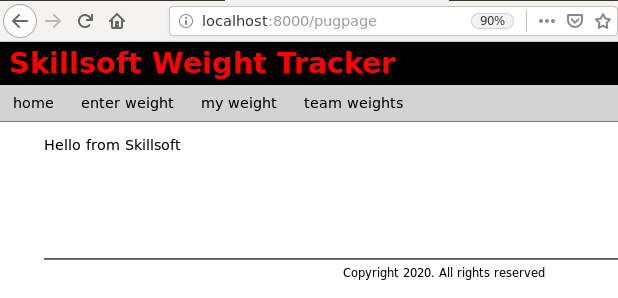
1. Change the controller.js function call to point to pughome.pug. Also remove the commented line.

|  |
| --- |
| **exports.pugpage=function(req, res){**  **res.render(**  **'pughome'**  **)**  **};** |

This will produce something like the same output that we had before, nothing changed even though we are pointing to a file that has just one line of content in it.

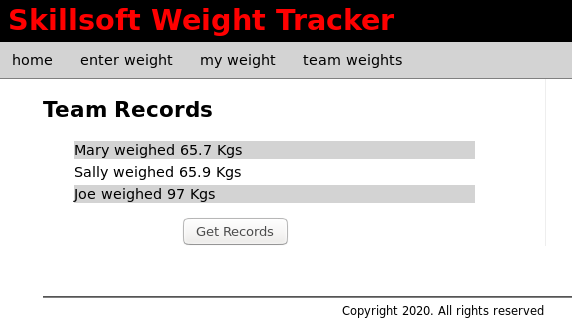
1. Remember the template had a block content area, this is where we insert new content for our new page. For example take a look at this next bit of code in the browser

|  |
| --- |
| **extends layout**  **block content**  **p Hello from Skillsoft** |

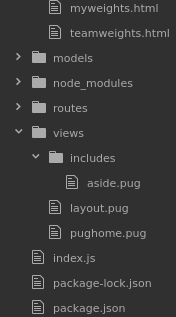
This would produce the following image  


1. Now all we have to do is build our page, but for this example we will simply borrow a page we already have, the teamweights.html page content. Since this page depends heavily on our API, include it in the appropriate places, both in the layout.pug file as well as any new pages we build.
2. First start building up the content of teamweights, copy the HTML between the <main> tags in teamweights.html.

|  |
| --- |
| **extends layout**  **block content**  **main**  **h2 Team Records**  **div#records**  **button(onClick="getData()") Get Records** |



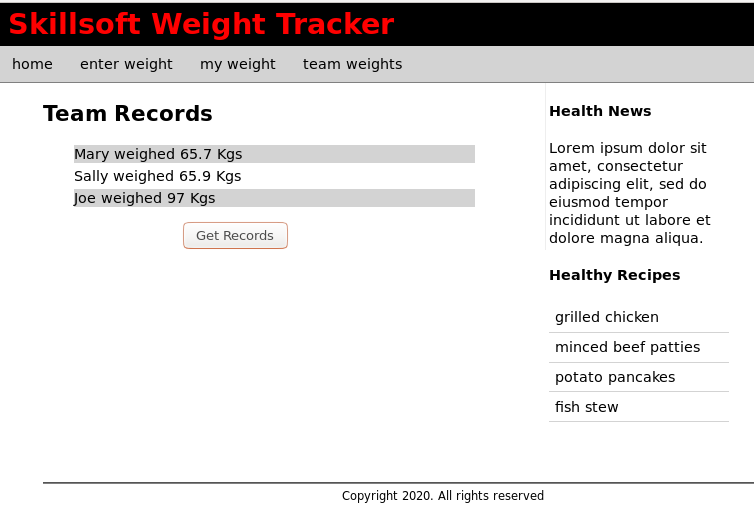
## part 7 – Including the Aside

1. Just like we built the layout and other web pages with templates we can also put the **aside** area of the page into a template, call it aside.pug. Usually though, it is better to create a folder called includes, then insert into that folder any file you wish to include at some point in time:

|  |
| --- |
| **aside**  **section**  **h4 Health News**  **p**  **| Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.**  **section**  **h4 Healthy Recipes**  **a(href='') grilled chicken**  **a(href='') minced beef patties**  **a(href='') potato pancakes**  **a(href='') fish stew** |

1. Now we can simply include the aside into our pug test page

|  |
| --- |
| **extends layout**  **block content**  **main**  **h2 Showing records for team**  **div#records**  **button#getData Get Records**  **include includes/aside** |



## part 8 – Install Morgan

1. Run the following command to install Morgan

**npm install morgan**

1. Create a folder called logs on the root of your app
2. Require morgan in the index.js file

|  |
| --- |
| **const express = require('express');**  **const bodyParser= require('body-parser');**  **const Morgan = require('morgan');**  **const port = 8000;**  **const app = express();**  **app.use(bodyParser.urlencoded({extended:false}));** |

1. Since Morgan requires access to the file system, lets include that as well in the main file

|  |
| --- |
| **const express = require('express');**  **const bodyParser= require('body-parser');**  **const Morgan = require('morgan');**  **const fs = require('fs');**  **const path = require('path');**  **const port = 8000;**  **const app = express();** |

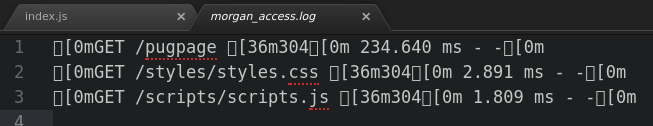
1. Now we can use the Morgan module to create a morgan object and create a log file. First create a folder called logs in the root of your application

|  |
| --- |
| **const Morgan = require('morgan');**  **const fs = require('fs');**  **const path = require('path');**  **const m = new Morgan(**  **'dev',**  **{stream:fs.createWriteStream('./logs/morgan\_access.log')}**  **);** |

Once you save these changes a log file will be created in the logs folder

1. Connect our app to morgan, but do this **after** the app object is created.

|  |
| --- |
| **const port = 8000;**  **const app = express();**  **app.use(bodyParser.urlencoded({extended:false}));**  **app.use(m);**  **const router = express.Router();** |

1. Run the application and go to the root folder or any path on the app we are building, then check the log file:  
     
   

## part 9 – Separate Morgan File

1. Create a new .js file inside of the controllers folder, call it morgan\_config.js
2. Add the following lines of code to this new file. Notice that it is the same lines in the index.js file, so you can just cut and paste:

|  |
| --- |
| **const fs = require('fs');**  **const path = require('path');**  **const Morgan = require('morgan');**  **//**  **const m = new Morgan(**  **'dev',**  **{stream:fs.createWriteStream('./logs/morgan\_access.log')}**  **);** |

1. In the morgan\_config.js file, add the exports line to make this function visible to other files

|  |
| --- |
| **const fs = require('fs');**  **const path = require('path');**  **const Morgan = require('morgan');**  **//**  **const m = new Morgan(**  **'dev',**  **{stream:fs.createWriteStream('./logs/morgan\_access.log')}**  **);**  **//**  **module.exports = m;** |

1. Now back in the index file, remove the corresponding lines

|  |
| --- |
| **const express = require('express');**  **const bodyParser= require('body-parser');**  **const port = 8000;**  **const app = express();**  **app.use(bodyParser.urlencoded({extended:false}));**  **const router = express.Router();**  **const routes = require('./routes/routes');**  **routes(app);**  **//**  **app.listen(port, () => console.log("Listening " + port));** |

1. In the index.js file, require the new morgan\_logs.js file

|  |
| --- |
| **const express = require('express');**  **const routes = require('./routes/routes');**  **const m = require('./controllers/morgan\_config');**  **const port = 8000;**  **const app = express();** |

1. Now we can use morgan as an object and pass it to app as middleware

|  |
| --- |
| **const m = require('./controllers/morgan\_config');**  **const port = 8000;**  **const app = express();**  **const router = express.Router();**  **routes(router);**  **//**  **app.use(m);**  **app.use(express.json());** |

1. This is the entire index.js file

|  |
| --- |
| **const express = require('express');**  **const routes = require('./routes/routes');**  **const m = require('./controllers/morgan\_config');**  **const port = 8000;**  **const app = express();**  **const router = express.Router();**  **routes(router);**  **//**  **app.use(m);**  **app.use(express.json());**  **app.use(express.urlencoded({extended:false}));**  **app.use('/', router);**  **app.use(express.static('HTML'));**  **app.set('view engine', 'pug');**  **//**  **app.listen(port, function(){**  **console.log("Listening " + port);**  **});** |

1. Test the app, it should function the same as before but now all logging details have been separated into a separate file

