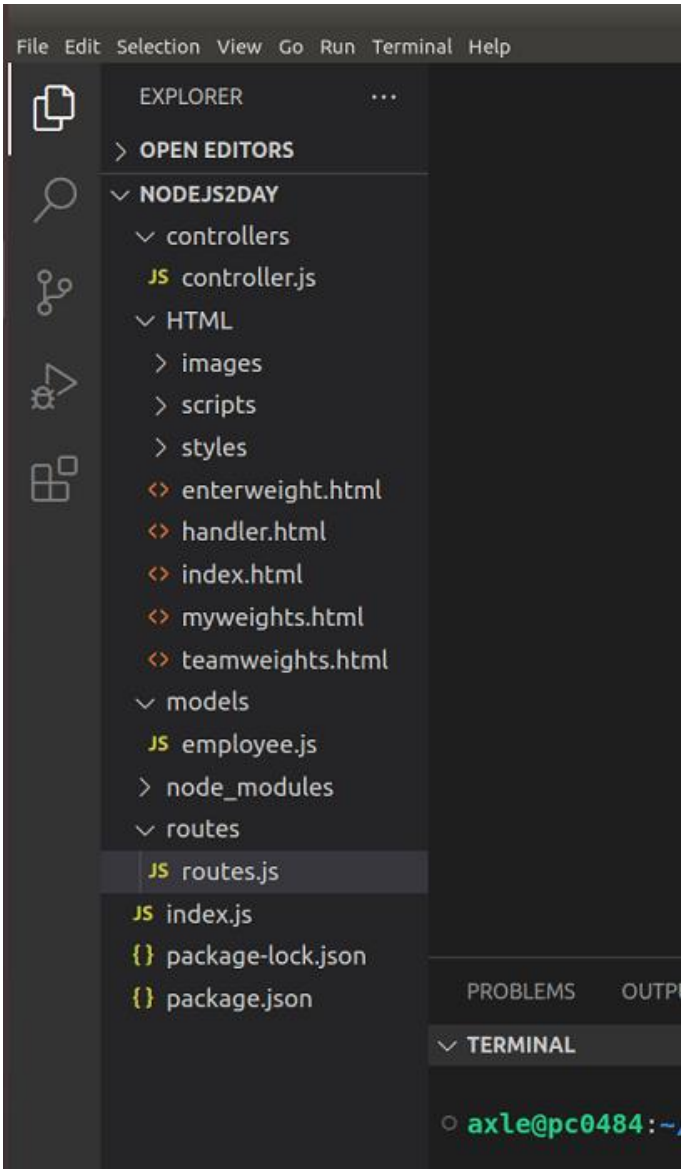
Table of Contents

Enhancing the site with HTML and JavaScript

This part of the course assumes that you understand the fundamentals of JavaScript. You are able to attach an external .js file to your HTML code and you are able to manipulate DOM elements (via their IDs or Names) using JavaScript.

You will be given starter files. The HTML files along with the .js and .css files represent a website created for a different project. We will use the HTML files here in this project.

SECTION 01 – CONNECTING TO THE APIS USING FETCH()



Note:

a. Your API must be running in order for your code in this section to work. If it is not running, go to the parent folder and run the nodemon command or **npm start.**

b. Also make sure your CORS plugin on the browser is turned on.

c. Since you are working here with the scripts.js file, remember to refresh your browser if you change this file, Nodemon does not know about scripts.js. If you are using VSCode and using a local server, this is not an issue.

d. The zipped file you are given contains all the HTML files we need to interact with our NodeJS API. Unzip that folder inside of the project folder you created on Day01. There should be two .html files and three folders.

1. We will be using mainly the allemployees.html file to connect to our back end API and display the data we have collected so far. Hook up this html file to our `.js` file. This just means adding this line just before the ending `</body>` tag:
<script src="scripts/scripts.js"></script>.

2. From the `main` div, remove the dummy text (if any) and just include a `div` to display the data from our database, and a button to call a function to get the data

```
<div id="container">
  <main>
    <h2>Employees in the Database</h2>
    <div id="documents"></div>
    <button onclick="getData();">Get Records</button>
  </main>
```

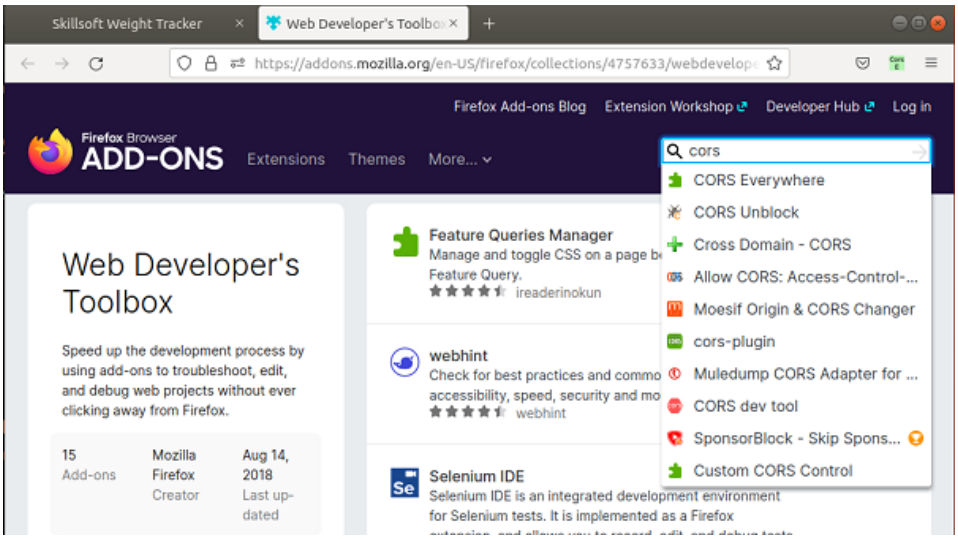
3. In the scripts.js file we can start writing the `getData()` function, put this code at the top of the document:

```
function getData(){
  fetch("http://localhost:8000/getemployees");
}
```

4. The `fetch()` returns an object, a **promise** object and the only way to handle that is with a `then()` method chained to the `fetch()` method. This may also be referred to as *subscribing* to the promise.

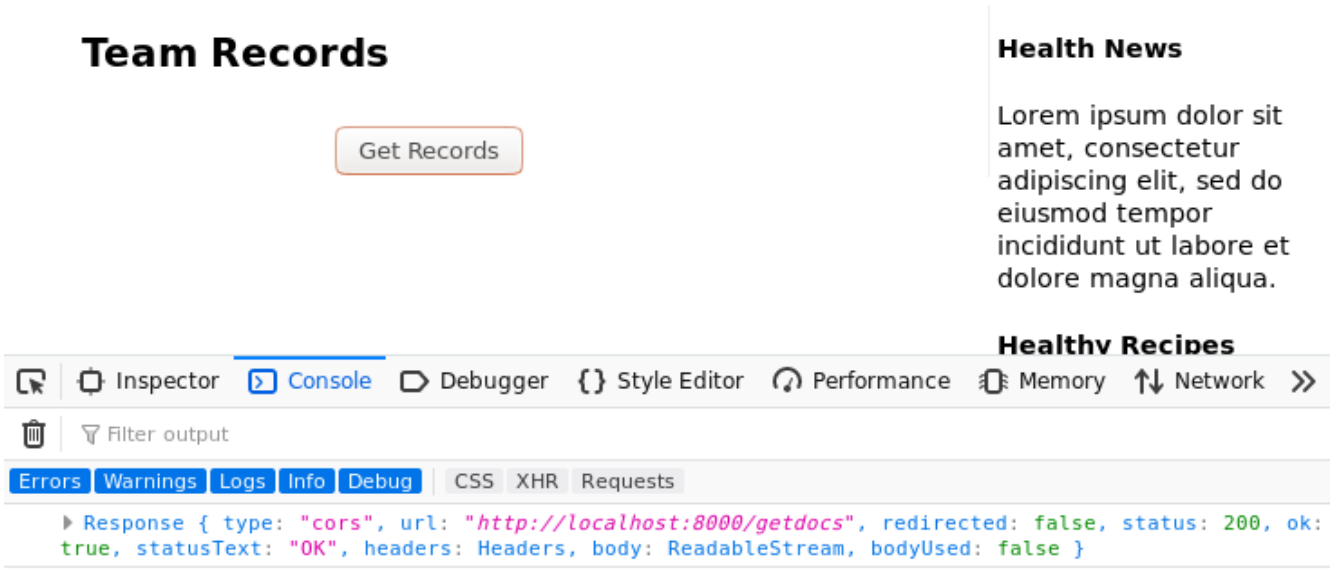
```
function getData(){
  fetch("http://localhost:8000/getemployees").then();
}
```

Note: It is at this point you may want to check that you have a CORS plugin. In my case with Mozilla Firefox, I am using **CORS Everywhere**. The image below shows how I search for it via Firefox's search feature and it is very easy to just add it to the browser. Once added to the browser, you can just click on it to turn it on or off



5. the **fetch()** method returns a Promise so we need a **then()** method to complete the transaction. Now within that **`then()`** method, you must supply a function that will handle any **`response`** from the **`fetch`** call. For now, we just log the response details:

```
function getData(){
    fetch("http://localhost:8000/getemployees").then(function(response){
            console.log(response);
    });
}
```



This is a lot of text to filter through. In order to extract the JSON body content from the response, we use the **json()** method. The *Request* and *Response* objects implements several methods like **text()** and **json()**.

Lets now add the *json* **parse()** method to the response and see what we get.

```
function getData(){
    fetch("http://localhost:8000/getdocs").then(function(response){
            console.log(response.json());
    });
}
```

This is much better, but it is still just a Promise object. Now we have no other option but to create a promise chain. We need to pass the value we receive from

the first Promise to a second **then()** method if we want to pull out data or perform further operations on the response.

6. So, instead of logging the response, let us return it

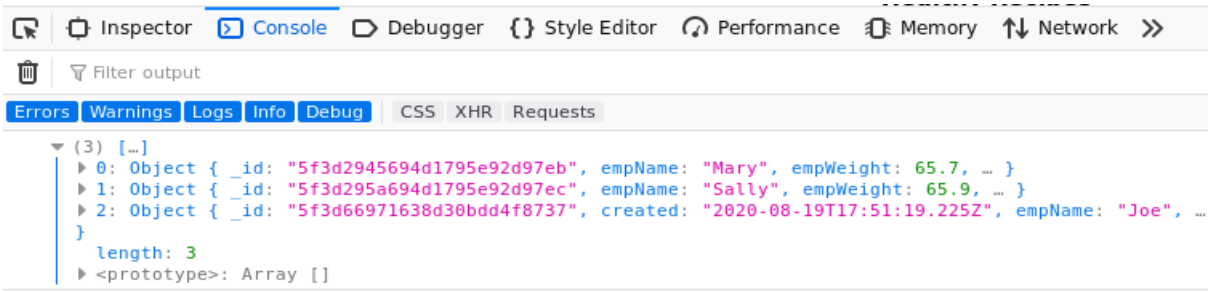
```
function getData(){
    fetch("http://localhost:8000/getemployees").then(function(response){
            return(response.json());
    });
}
```

7. But now it means we need another **then()** method

```
function getData(){
    fetch("http://localhost:8000/getemployees").then(function(response){
            return response.json()).then();
    });
}
```

8. The second then method also takes a function, and it expects data, which we can log for now

```
function getData(){
  fetch("http://localhost:8000/getemployees").then(function(response){
    return(response.json()).then(function(data){
      console.log(data);
    });
  })
}
```



Finally, we have the data we were looking for.

9. Usually though it is better to write the code in a more structured way:

```
function getData(){
  fetch("http://localhost:8000/getemployees")
    .then(function(response){
      return(response.json())
    .then(function(data){
      console.log(data);
    });
  })
};
```

10. This way we can complete the **`getData()`** function by also inserting a **`catch()`** method. This is just in case anything went wrong. In this way we say that the **`catch()`** method is *chained* to the **`then()`** method which is *chained* to the **`fetch()`** method.

```
function getData(){
  fetch("http://localhost:8000/getemployees")
    .then(function(response){
      return(response.json())
    .then(function(data){
      console.log(data);
    }).catch(function(err){
      console.log(err);
    });
  })
};
```

11. Using arrow functions

```
function getData(){
  fetch("http://localhost:8000/getemployees")
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(err => console.log(err))
};
```

SECTION 02 – DISPLAY THE DATA

1. Remember we had a **`div`** tag in the allemployees.html file that we can use to display the data, this **`div`** has an **`id`** of **`documents`**. We will use this tag and some DOM manipulation to display the data.

2. In the scripts.js file add a new function just beneath the **`getData()`** function, called **`displayData()`**

```
function displayData(arr) {
  const container = document.getElementById("documents");
}
```

We also need to get access to the **`documents`** div tag on the HTML page hence **`getElementsById()`**.

3. The data in the console showed up as an array so we need an array structure to get the data out. A normal loop will do here. For each *document* in the array, we add a new **list item** (or div):

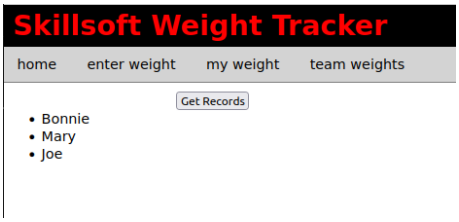
```
function displayData(arr) {
  const container = document.getElementById("documents");
  for (let i = 0; i < arr.length; i++) {
    const li_employee = document.createElement('li');
  }
}
```

4. Now we can get the data from the array and add it the list item from #3. The next step in this part is to add the list item the container from #2.

```
function displayData(arr) {
      const container = document.getElementById("documents");
  for (let i = 0; i < arr.length; i++) {
    const li_employee = document.createElement('li');
    li_employee.innerHTML = arr[i].empName;
    container.appendChild(li_employee);
  }
}
```

5. Now instead of logging the data, pass it, as an array, to **displayData():**

```
function displayData(arr) {
  const container = document.getElementById("documents");
  for (let i = 0; i < arr.length; i++) {
      const li_employee = document.createElement('li');
      li_employee.innerHTML = arr[i].empName;
      container.appendChild(li_employee);
    }
}
```



Here are the two functions so far :

```
function getData(){
  fetch("http://localhost:8000/getemployees")
  .then(response => response.json())
  .then(data => displayData(data))
  .catch(err => console.log(err))
}
//
function displayData(arr) {
  const container = document.getElementById("documents");
  for (let i = 0; i < arr.length; i++) {
    const li_employee = document.createElement('li');
    li_employee.innerHTML = arr[i].empName;
    container.appendChild(li_employee);
  }
}
```

Refer to Appendix E to see the same code effect using the Async and Await construction instead of then() and catch()

We will use one of the HTML files given in the set of starter files. Look for the addemployee.html file and we will configure it to pass data from the enclosed form into the database. We will ignore several security issues for this bootcamp, such as validation and encryption.

1. Our database at the moment can handle two fields, *empName* and *empPass*, both are simple and are string fields. Change the **id** and **name** fields on the HTML so that these fields reflect the proper naming as defined in the database.

2. The *form* tag, at the moment, just has an *id* of *signup* and a method, *post*. Also the HTML file itself is connected to the scripts.js file via the usual linking at the bottom of the document. If this *script* tag is not there, add it now:

```
        </footer>
        <script src="scripts/scripts.js"></script>
    </body>
    </html>
```

3. There are several ways to submit the form fields and values to the server running on localhost. In this method we will *listen* for the button click on the form, then use the **fetch()** method to POST the values entered by the user. First at the top of the .js file, add a variable to represent the form itself. Then later down use the **addEventListener()** method that is automatically part of the form and configure it as shown:

```
const userForm = document.getElementById("signup");

...other code here

userForm.addEventListener("submit", (e) => {
    e.preventDefault();
});
```

We are listening for the *submit* event and when it happens, the event along with the object that caused that event will be captured in the variable **e**. The **preventDefault()** is part of the HTML specification and it will prevent the form from being submitted by mistake by the user.

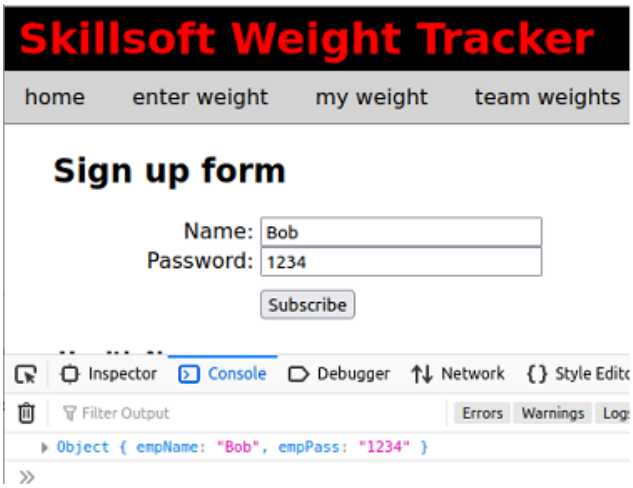
4. The next two lines will first get a handle to the form itself and then use the JavaScript **FormData()** method to extract the two fields into an object:

```
userForm.addEventListener("submit", (e) => {
    e.preventDefault();
    let form = e.currentTarget;
    let formFields = new FormData(form);
```

5. The funciton **FormData()** by itself is not enough to wrap values. We will use the modern *Object.formEntries* to gather up all the values the user enters via those fields:

```
let form = e.currentTarget;
let formFields = new FormData(form);
let formDataObject = Object.fromEntries(formFields.entries());
```

At this point if you log the **formDataObject** you will see the form already wrapped up with field/value pairs.



Note: Object.formEntries is available by default in your Browser. Check specification ECMAScript 2017 for more details.

1. Now that we have a neat little object all wrapped up and ready to go, we can now use the same **fetch()** method to POST this little object to our back end, specifically to the *addemployee* endpoint.

```
        let formDataObject = Object.fromEntries(formFields.entries());
        fetch('http://localhost:8000/addemployee', {});
    })
```

As you can see the fetch method takes a second parameter. That parameter is an object and it can be configured to pass information to the server, it is empty at the moment.

2. That second parameter can itself accept several configuration details, for now we only need three, the *method*, *headers* and a *body*:

```
        fetch('http://localhost:8000/addemployee', {
                method: ,
                headers: {
                },
                body:
        });
});
```

3. The *method* in this case is POST, the *headers* is simply telling the server that we are sending JSON data and finally the *body* is the actual form fields and values wrapped up into a neat object for our back end API:

```
        fetch('http://localhost:8000/addemployee', {
                method: 'POST',
                headers: {
                        'Content-Type': 'application/json',
                },
                body: JSON.stringify(formDataObject),
        });
```

Notice that the *headers* is an object. We use the right side to inform the server that we are sending JSON content. Finally we wrap up the form object into JSON using the stringify() method of JSON. JSON is also part of your browser.

4. At this point we have everything we need but the form will never get submitted even if we hit the subscribe button on the HTML form. The reason is that the **fetch()** method returns a Promise object and unless you handle the Promise in the proper way, the values submitted simply will get lost and never reach the server. What we have to do is attach a **.then()** method to our *fetch* and then the form will get submitted:

```
        },
                body: JSON.stringify(formDataObject),
        })
        .then();
});
```

5. Although this will work, it is better to add a few more details. For example, if the server responds with data, you need to be able to capture that data.

```
                body: JSON.stringify(formDataObject),
        })
        .then(function(response){
                console.log(response);
        });
});
```

Note: if the server responds with JSON data, this may not work, it all depends on what is being sent back by the server. Also, this is a good point to log that response using a more developed logging service such as Winston or Log4JS.

6. Finally, we need to add a **catch()** method to capture and log any errors that may occur:

```
                body: JSON.stringify(formDataObject),
        })
        .then(function(response){
                console.log(response);
        }).catch(function(err){
                console.log(err);
        });
});
```

SECTION 05 – 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`**

2. Also import the **`jsonwebtoken`** package at the top of controller.js

```
const jwt = require('jsonwebtoken');
const Employee = require('../models/employee');
```

3. In controllers.js file, copy the **`addemployee`** 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;

};
```

4. 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;
    Employee.find({ empName: empName })

};
```

As you remember from the other find() methods, we now have to handle them asynchronously, so we need a then() method to start with.

5. If we supply the then() method, then if we supply a parameter, employeeData in this case, we can capture whatever the database responds with:

```
Employee.find({ empName: empName })
  .then(
    employeeData => {
      //check that we have something, if not, send error message
      if (employeeData.length === 0)
        res.send({ "message": empName + " not found!" });
      else {
        //we have an object, so test the password
        if (employeeData[0].empPass == empPass) {
          //see if both passwords match
        }
      }
    }
  )
  .catch((err) => {
    //error in waiting for employeeData
    res.send(err);
  })
```

6. Next step is to check to call the sign() method of the jwt object. I also added an else clause since if the passwords don't match, we have an invalid user:

```
Employee.find({ empName: empName })
  .then(
    employeeData => {
      //check that we have something, if not, send error message
      if (employeeData.length === 0)
        res.send({ "message": empName + " not found!" });
      else {
        //we have an object, so test the password
        if (employeeData[0].empPass == empPass) {
          //see if both passwords match
          var token = jwt.sign();
        } else {
          res.end("Login Failed")
        }
      }
    }
  )
```

7. The sign() method of the jwt object takes a minimum of 3 things, an object called the payload, a string that works like a key and a callback function that contains the token or an error. I have added a timeout object also as the fourth:

```
        else {
         //we have an object, so test the password
         if (employeeData[0].empPass == empPass) {
           //see if both passwords match
           var token = jwt.sign(
             {

           },
           "",
           {  },
           (err, token) => {

           }
         );
        } else {
          res.end("Login Failed")
        }
           }
```

8. Here I added in the details for each part. The payload can be any object, here I am adding the employee's name and user id. The key can be any string, and the expiry can be any time frame or be forever. Finally the callback function is the most important, it contains the actual token in the **token** variable here:

```
            if (employeeData[0].empPass == empPass) {
              //see if both passwords match
              var token = jwt.sign(
                {
                  empName: employeeData[0].empname,
                  userID: employeeData[0]._id
                },
                "shhhh",
                { expiresIn: "1h" },
                (err, token) => {
                  if (err) res.send(err);
                  res.send(token);
                }
              );
```

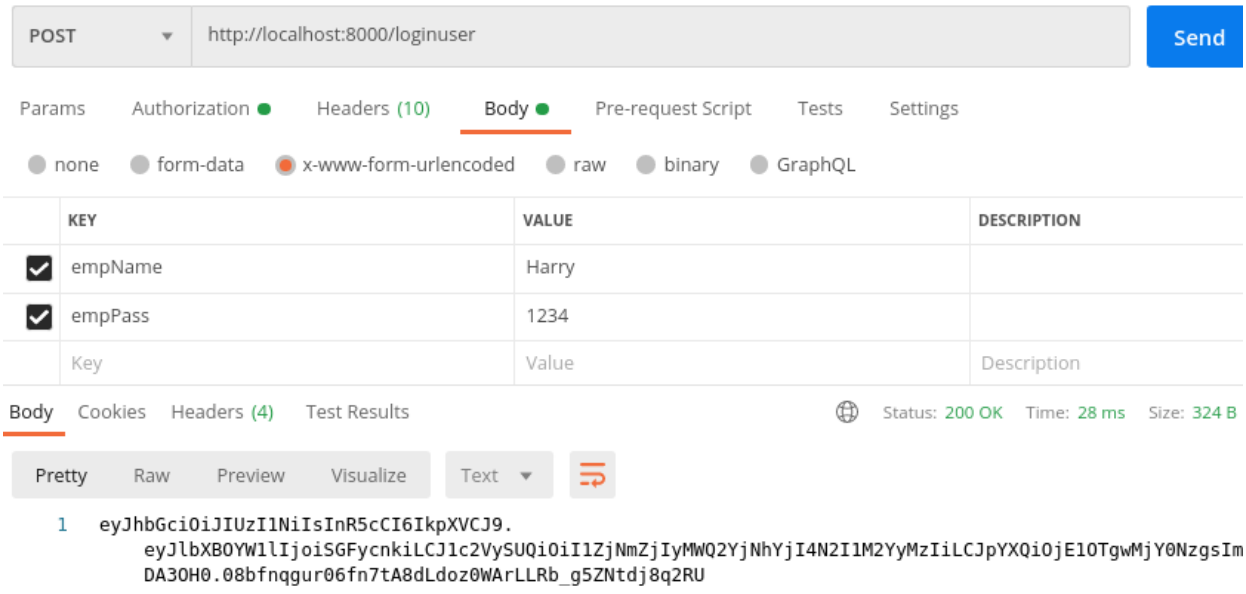
9. In routes.js file, add routes to handle user login, the controller function already exist. Make sure that they are POST routes:

```
        router.post('/addemployee', controller.addemployee);
        router.put('/updateemployee', controller.updateemployee);
        router.post('/loginuser', controller.loginuser);
    }
```

10. Here is the entire loginuser() function:

```
exports.loginuser=function(req, res){
 let empName = req.body.empName;
 let empPass = req.body.empPass;
 Employee.find({ empName: empName })
  .then(
   employeeData => {
    if (employeeData.length === 0)
     res.send({ "message": empName + " not found!" });
    else {
     if (employeeData[0].empPass == empPass) {
      var token = jwt.sign(
       {
        empName: employeeData[0].empname,
        userID: employeeData[0]._id
       },
       "shhhh",
       { expiresIn: "1h" },
       (err, token) => {
        if (err) res.send(err);
        res.send(token);
       }
      );
     } else {
      res.end("Login Failed")
     }
    }
   }
  )
  .catch((err) => {
   res.send(err);
  })
};
```

11. Lets sign in a 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:

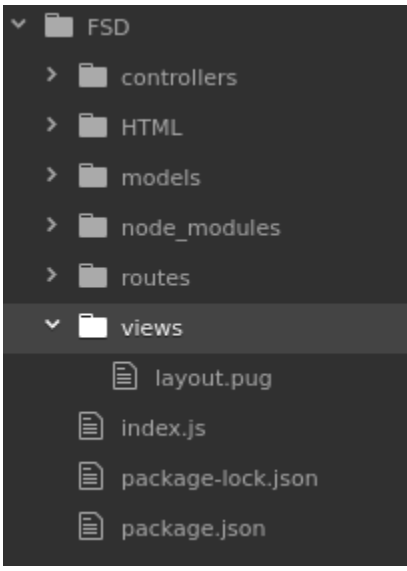


Note: if you want to see the other side of tokens refer to **Appendix G**.

1. Run the following command to install Pug (remember to stop the application first)
   `npm install pug --save`

2. 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(express.urlencoded({extended:false}));
app.set('view engine', 'pug');
const router = express.Router();
const routes = require('./routes/routes');
```



3. 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)

4. 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
```

5. Over in the controllers.js file, add a method to handle this Pug test route, just copy the **`aboutus`** function and comment the code that's already there:

```
//
exports.pughome=function(req, res){
  // res.send('You are on the pug home route.');
};
```

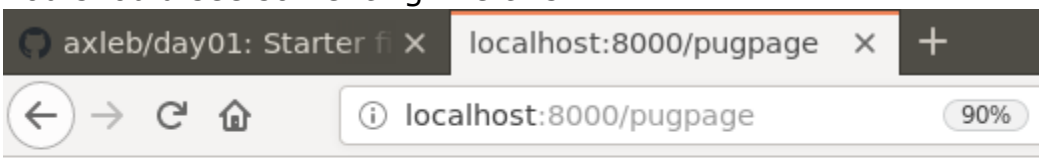
6. Then create a new route in routes.js to point to a pug test page

```
    router.put('/updateemployee', controller.updateemployee);
    router.post('/loginuser', controller.loginuser);
    router.get('/pughome', controller.pughome);
};
```

7. Now back in the new controller function, simply call the **render()** method from *res*, 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



- home

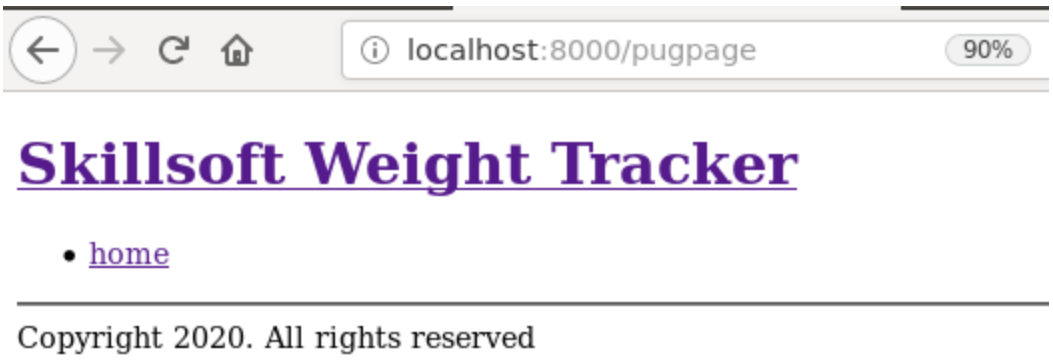
8. Add some content so that we can test this layout.pug file

```
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 2023. All rights reserved
```



SECTION 07 – 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 2023. All rights reserved
    script(src='scripts/scripts.js')
```

   Test the layout route again.

2. At this point, you may notice quite a few errors in the console window of your browser. These errors do not show up on the page but they could be an issue later on in the development process. To fix these errors we have to let our Node app know that we are working with static HTML files. Add this line to your index.js file:

```
const router = express.Router();
routes(router);
app.use(express.static('HTML'));
app.use(express.json());
app.use(express.urlencoded({extended:false}));
```

3. In order to demonstrate how this Pug 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. When we create new pages from now on, we simply include the wrapper or layout template by extending it.

```
extends layout
```

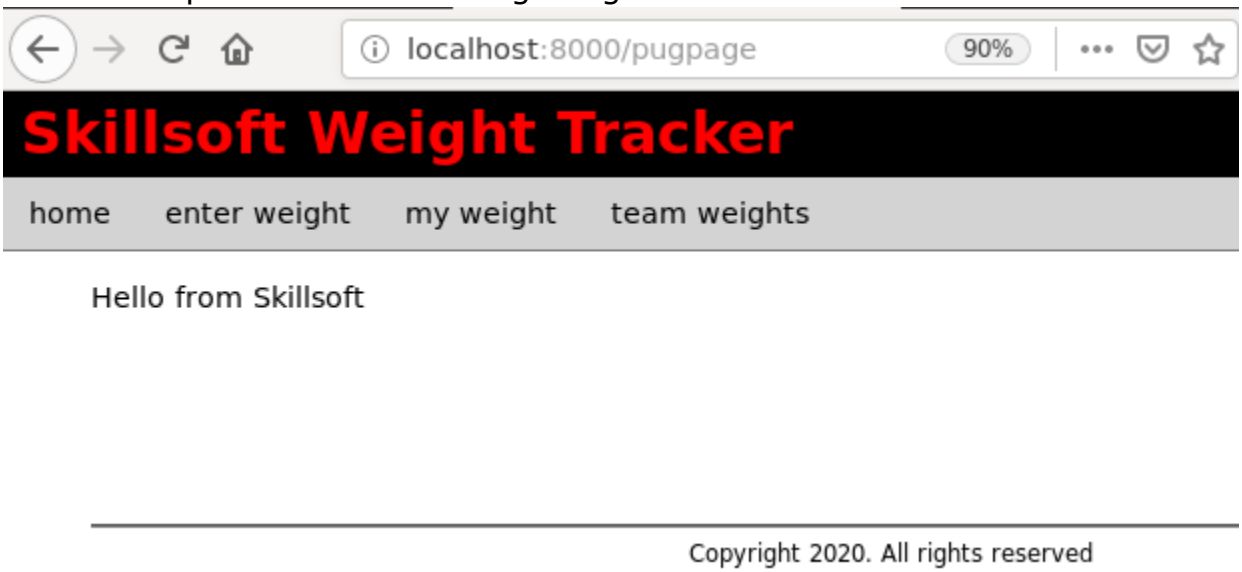
After doing this, change the **pughome()** function in controller.js to render *pughome* instead of *layout*

4. 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

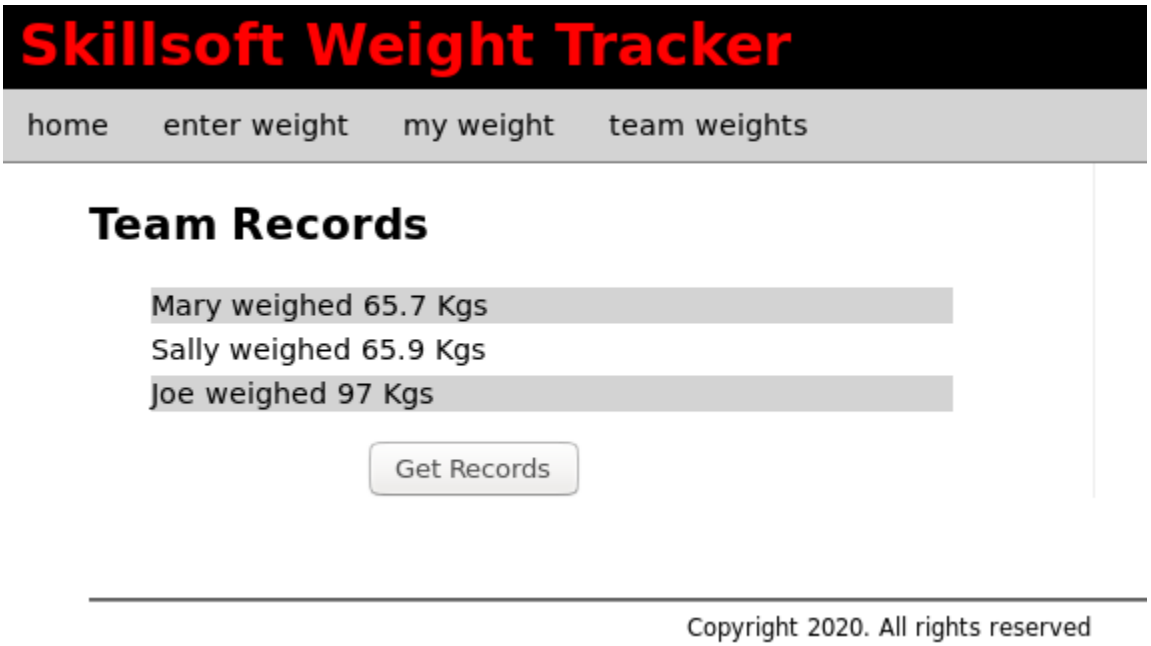


5. Now all we have to do is build our page, but for this example we will simply borrow a page we already have, the allemployees.html page content. This page depends heavily on our API, so it's a good page to use here to demonstrate Pug.

6. First start building up the content of pughome based on the code in dbdump, copy the HTML between the <main> tags but just main without the <>:

```
extends layout

block content
  main
    h2 Team Records
    div#documents
    button(onClick="getData()") Get Records
```



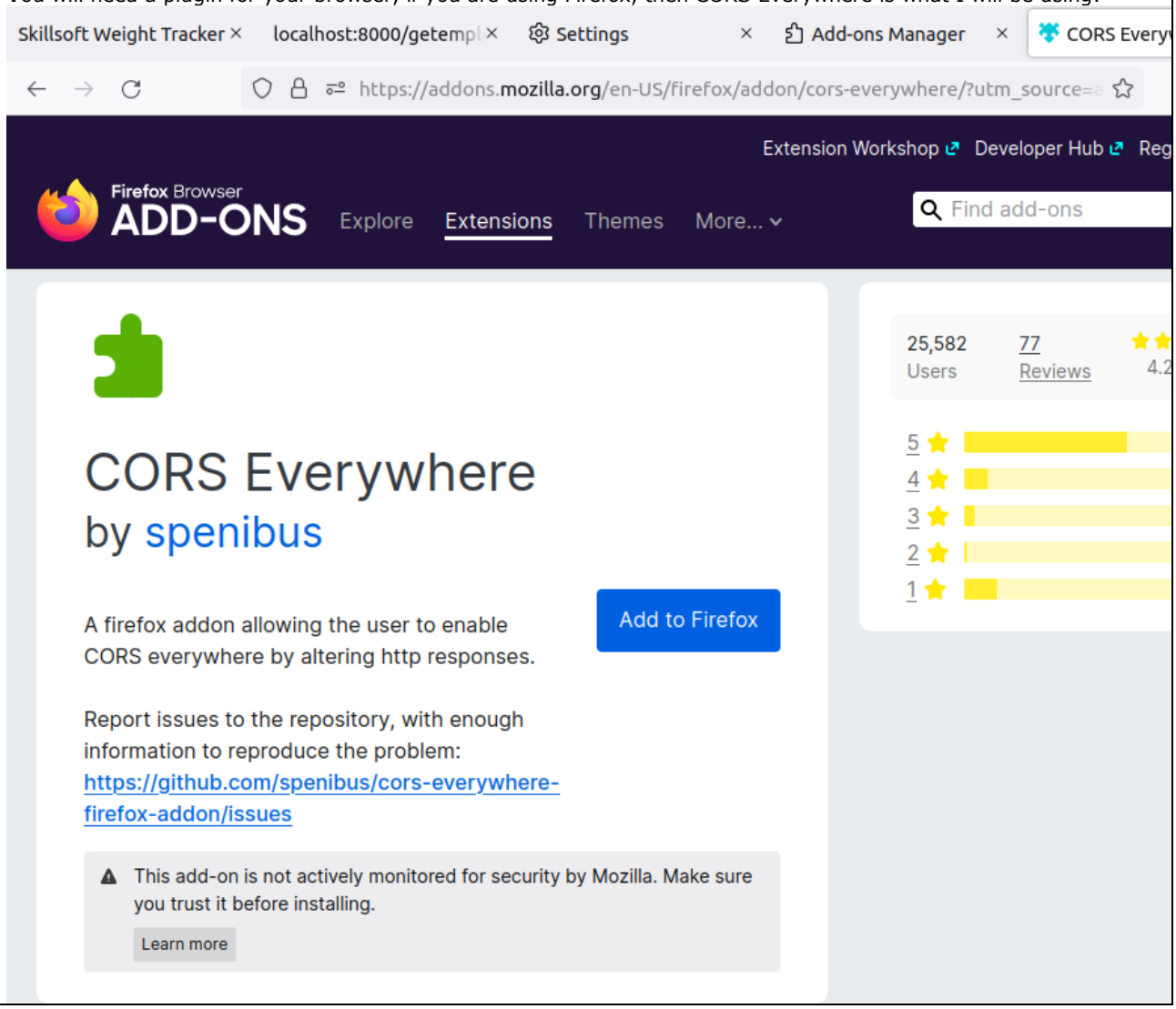
Note, if you want to see the entire page, complete **Appendix F**. In that appendix I added in the *aside* part of the web page.

Appendix A – Using the Map Method to Display Data

```
function displayData(arr) {
  document.getElementById("records").innerHTML = arr.map(mapOutput).join("");
}
function mapOutput(emp){
  outHTML = emp.empName + " weighed " + emp.empWeight + "<br />";
  return outHTML;
}
```

Appendix B – CORS Plugin

You will need a plugin for your browser, if you are using Firefox, then CORS Everywhere is what I will be using:



APPENDIX C – DISPLAY THE DATA (OLD SCHOOL BUT SIMPLE)

1. Remember we had a **`div`** tag in the allemployees.html file that we can use to display the data, this **`div`** has an **`id`** of **`documents`**. We will use the innerHTML of this tag to display the data.

2. In the scripts.js file add a new function just beneath the **`getData()`** function, called **`displayData()`**

```
function displayData(arr) {
    let outHTML = "";

    document.getElementById("documents").innerHTML = outHTML;
}
```

   Notice that **`outHTML`** is a new variable which we will use to append records as we iterate through the array containing our data lines.

3. The data in the console showed up as an array so we need an array structure to get the data out

```
function displayData(arr) {
    let outHTML = "";
    for(let i=0; i < arr.length; i++){
            outHTML+="<p>"+arr[i].empName + " using password " + arr[i].empPass + "</p>";
    }
    document.getElementById("documents").innerHTML = outHTML;
}
```

4. Now call this **`displayData()`** function from the **`getData()`** function, via its **`then()`** method.

```
function getData(){
  fetch("http://localhost:8000/getemployees")
    .then(response => response.json())
    .then(data => displayData(data))
    .catch(err => console.log(err))
};
```

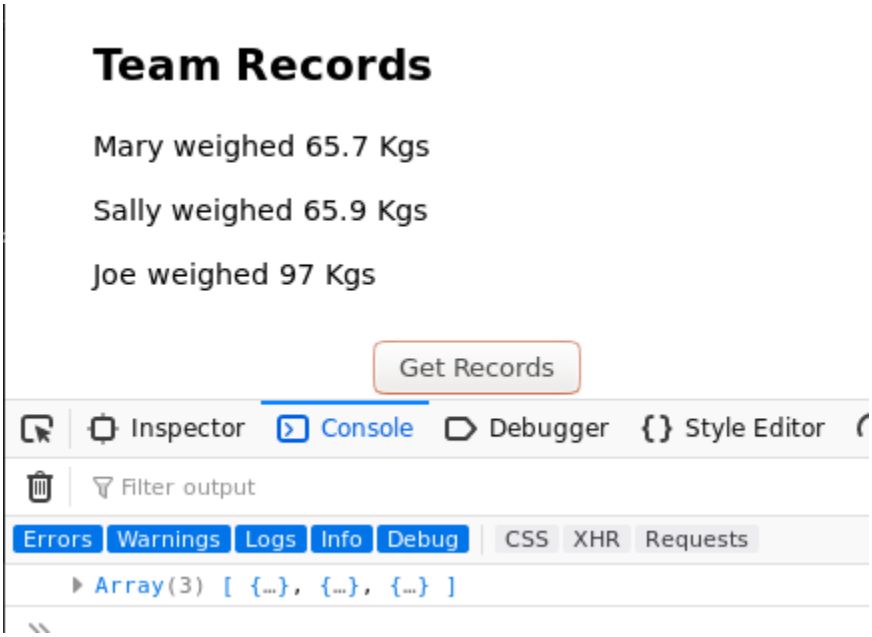
5. What if you also wanted to log the data. As it turns out we could have multiple **`then()`** methods in a structure. But because we already used the **`response`** within a function, we would have to manually pass it to the next chained event by using a **`return`** statement. In this case we would need {} for our second then() mehtod.

```
function getData(){
  fetch("http://localhost:8000/getemployees")
    .then(response => response.json())
    .then(data => {
      displayData(data);
      return(data);
    })
    .catch(err => console.log(err))
};
```

6. Now that we are returning data, lets now create a third then() method to process this new line of code.

```
function getData(){
  fetch("http://localhost:8000/getemployees")
    .then(response => response.json())
    .then(data => {
      displayData(data);
      return(data);
    })
    .then(data=>console.log(data))
    .catch(err => console.log(err))
};
```

Now we have the data displayed AND we have the same returned data being displayed in the console window, so we were able to use one promise object two times



Here are the two functions so far using fetch()

```
function getData(){
  fetch("http://localhost:8000/getemployees")
    .then(function(response){
      return(response.json())
    .then(function(data){
      displayData(data);
    });
  }) ;
};
//
function displayData(arr) {
      let outHTML = "";
      for(let i=0; i < arr.length; i++){
            outHTML+="<p>"+arr[i].empName + " using password " + arr[i].empPass + "</p>";
      }
      document.getElementById("documents").innerHTML = outHTML;
}
```

APPENDIX D – ASYNC OPTION FOR POSTING DATA

1. If you were looking for an async option to post data, well, it's a bit more difficult. It is possible to make the addEventListener() accept an async function, but the code is more complicated. This is a reasonable compromise:

```
userForm.addEventListener("submit", (e) => {
    e.preventDefault();
 let form = e.currentTarget;
    let formFields = new FormData(form);
 let formDataObject = Object.fromEntries(formFields.entries());
 addNewEmployee(formDataObject)
 .then(status => {
  console.log(status);
 });
});

async function addNewEmployee(formDataObject) {
  const response = await     fetch('http://localhost:8000/addemployee', {
                method: 'POST',
                headers: {
                        'Content-Type': 'application/json',
                },
    body: JSON.stringify(formDataObject),
     });
  if (!response.ok) {
   const message = `An error has occured: ${response.status}`;
   throw new Error(message);
  }
  const allGood = await response.json();
  return allGood;
 }
```

APPENDIX E – USING ASYNC/AWAIT

Section 3 above can be re-written with the **`Async/Await`** structure. It is newer than **`then()/catch()`** but you will need to add in your own error handling if using this structure.

In order to use the **`async/await`** sturcture, we first have to make the **`getData()`** function an **`async`** function. After that we **`await`** the results of a **`fetch()`** operation which just like before returns a **`response`** object. We would need to apply **`await`** again in order to extract the json object from the response object.

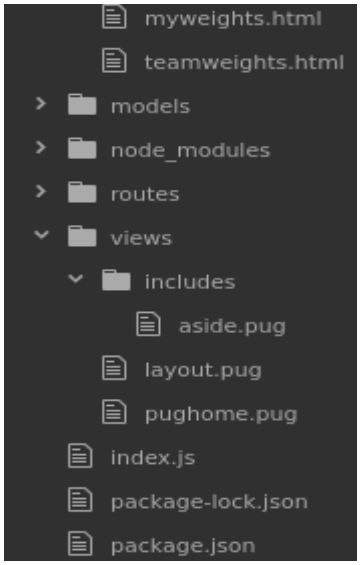
```
async function getData(){
  const response = await fetch("http://localhost:8000/getemployees");
  const data = await response.json();
  displayData(data);
};
```

With error handling:

```
async function getData(){
  try{
    const response = await fetch("http://localhost:8000/getemployees");
    const data = await response.json();
    displayData(data);
  } catch(err){
    console.log(err);
  }
};
```

APPENDIX F – INCLUDING THE ASIDE USING PUG

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:

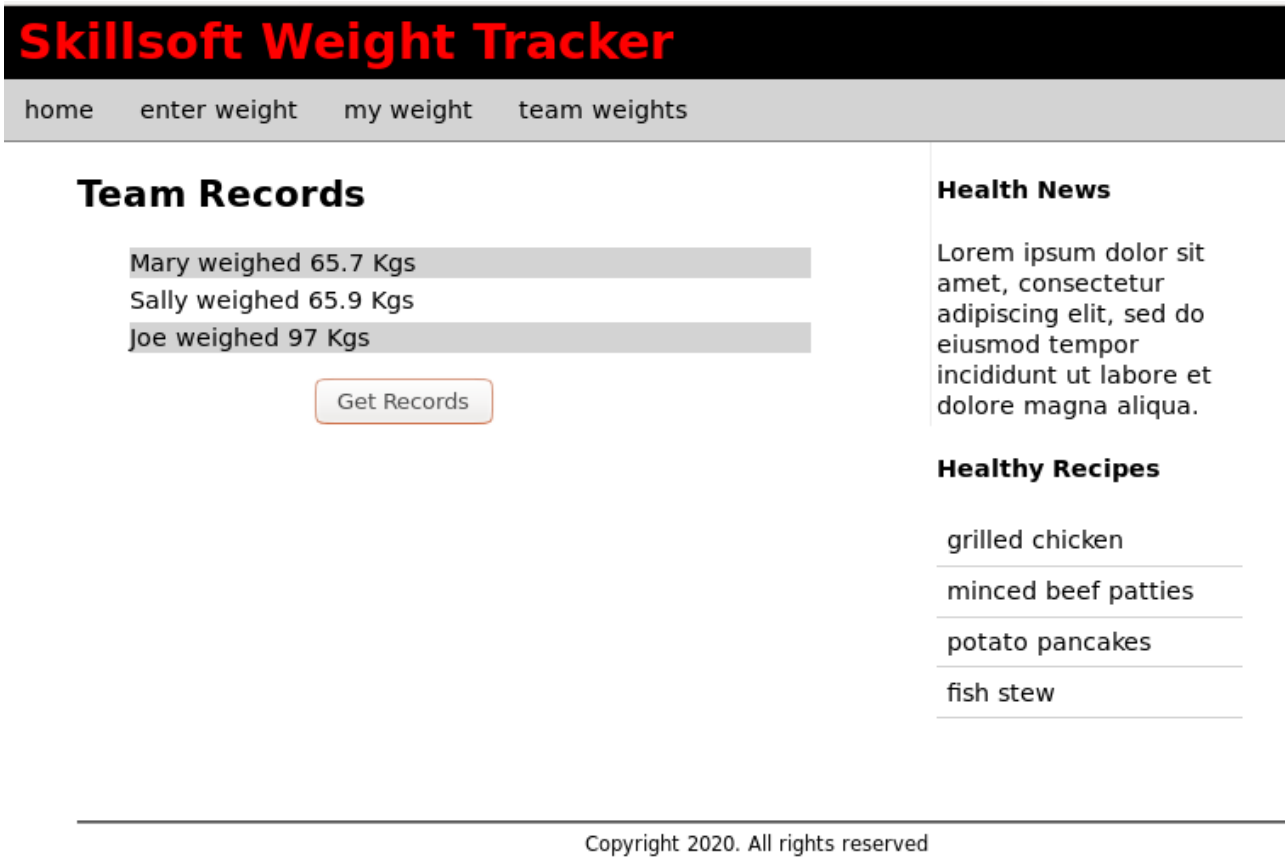



```
aside
  section
    h4 Health News
    p
      | Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore   et dolore magna aliqua.
  section
    h4 Healthy Recipes
    a(href='') grilled chicken
    a(href='') minced beef patties
    a(href='') potato pancakes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

2. 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
```



APPENDIX G – 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) {


}
```

2. 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;

}
```

3. 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.

4. 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');
  }

}
```

5. 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();
  }
```

6. 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"});
 }
}
```

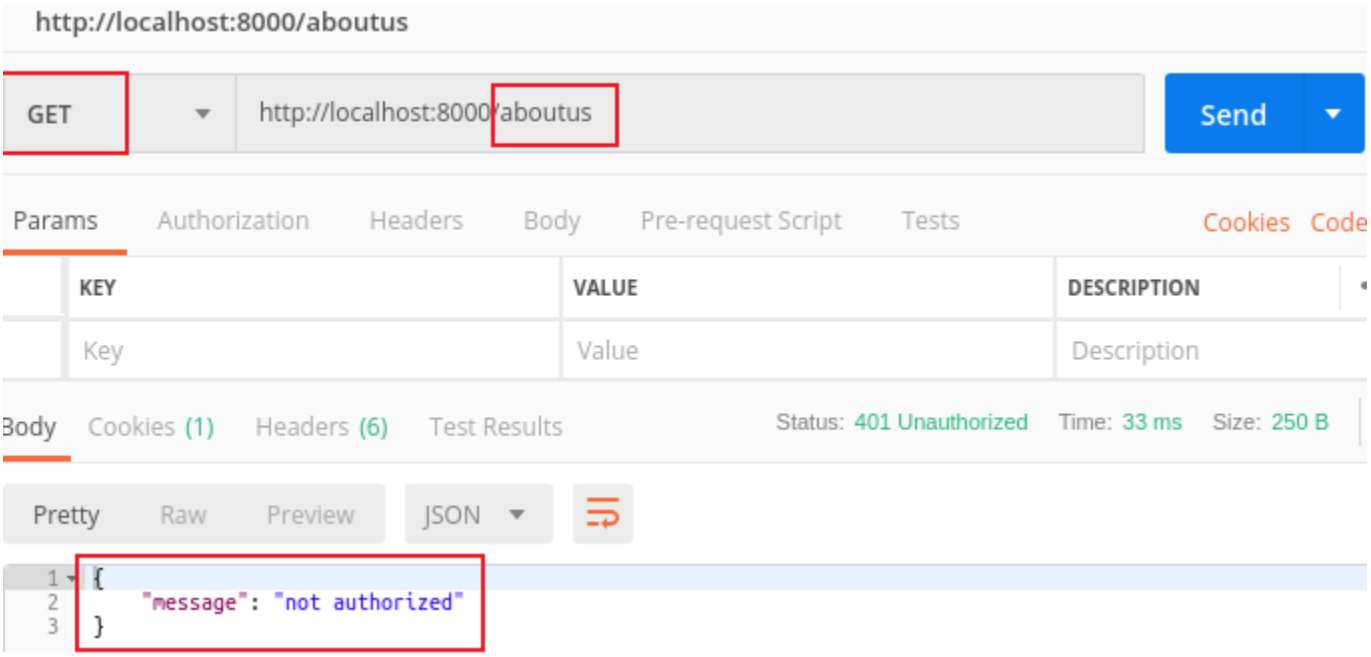
7. All that's left now is to protect a route, first import the auth.js file we just created into the routes.js file

```
const controller = require('../controllers/controller');
let authUser = require('../controllers/auth');
module.exports = function(router){
 //
 router.get('/', controller.getdefault);
```

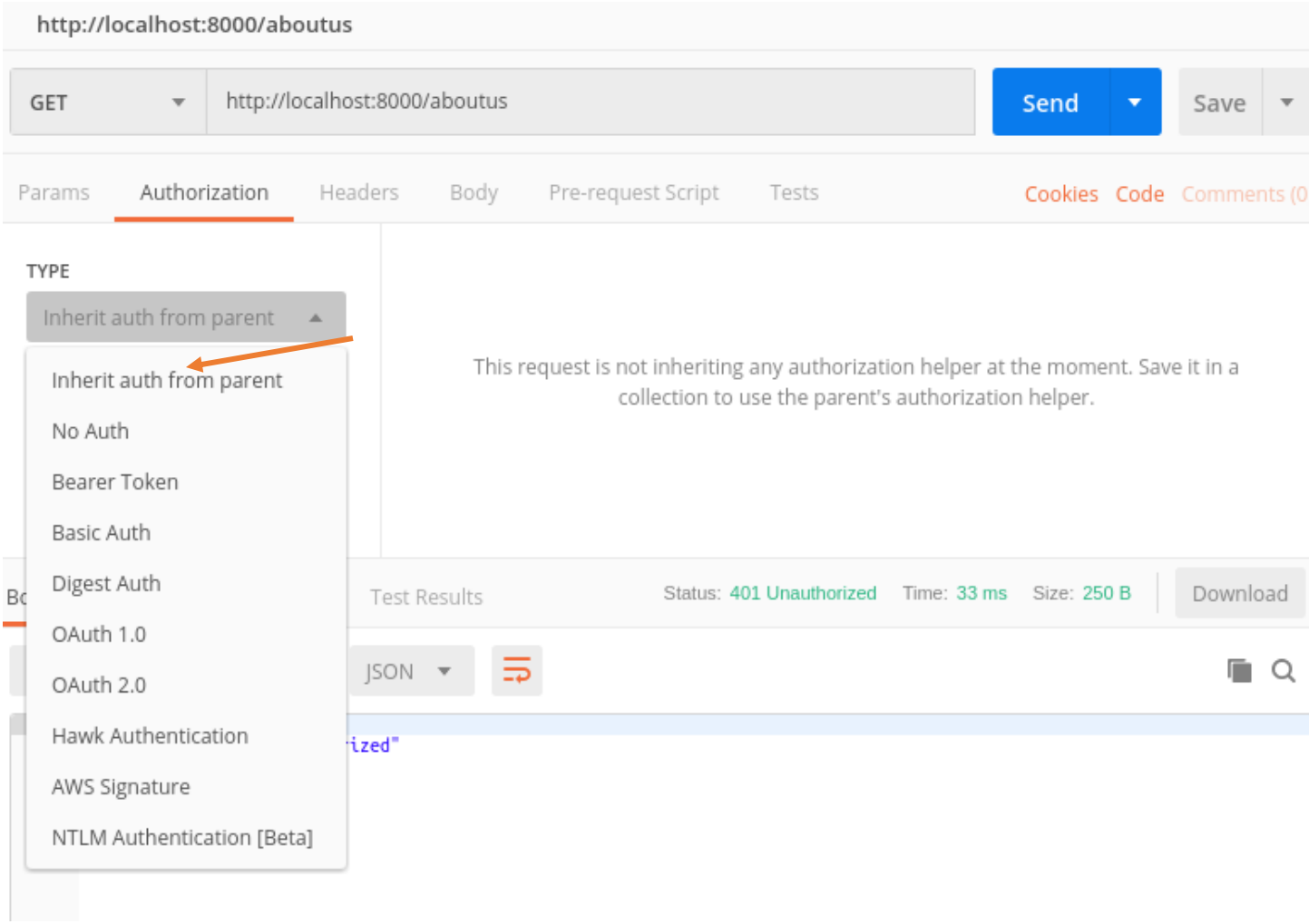
8. We will experiment with the aboutus route, in terms of protecting this route. Simply insert the authUser variable before the controller part

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

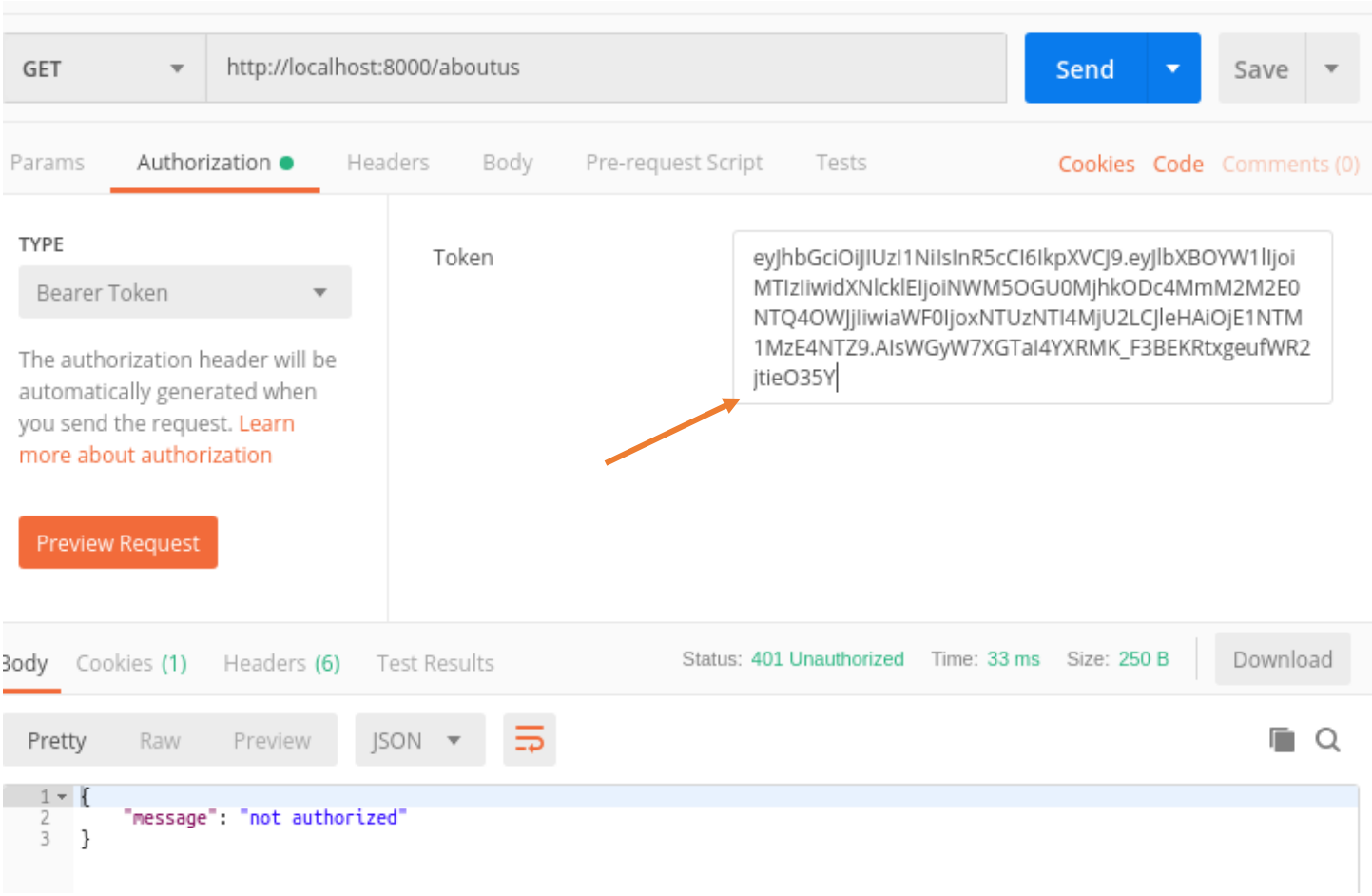
9. Now we can test first without the token:



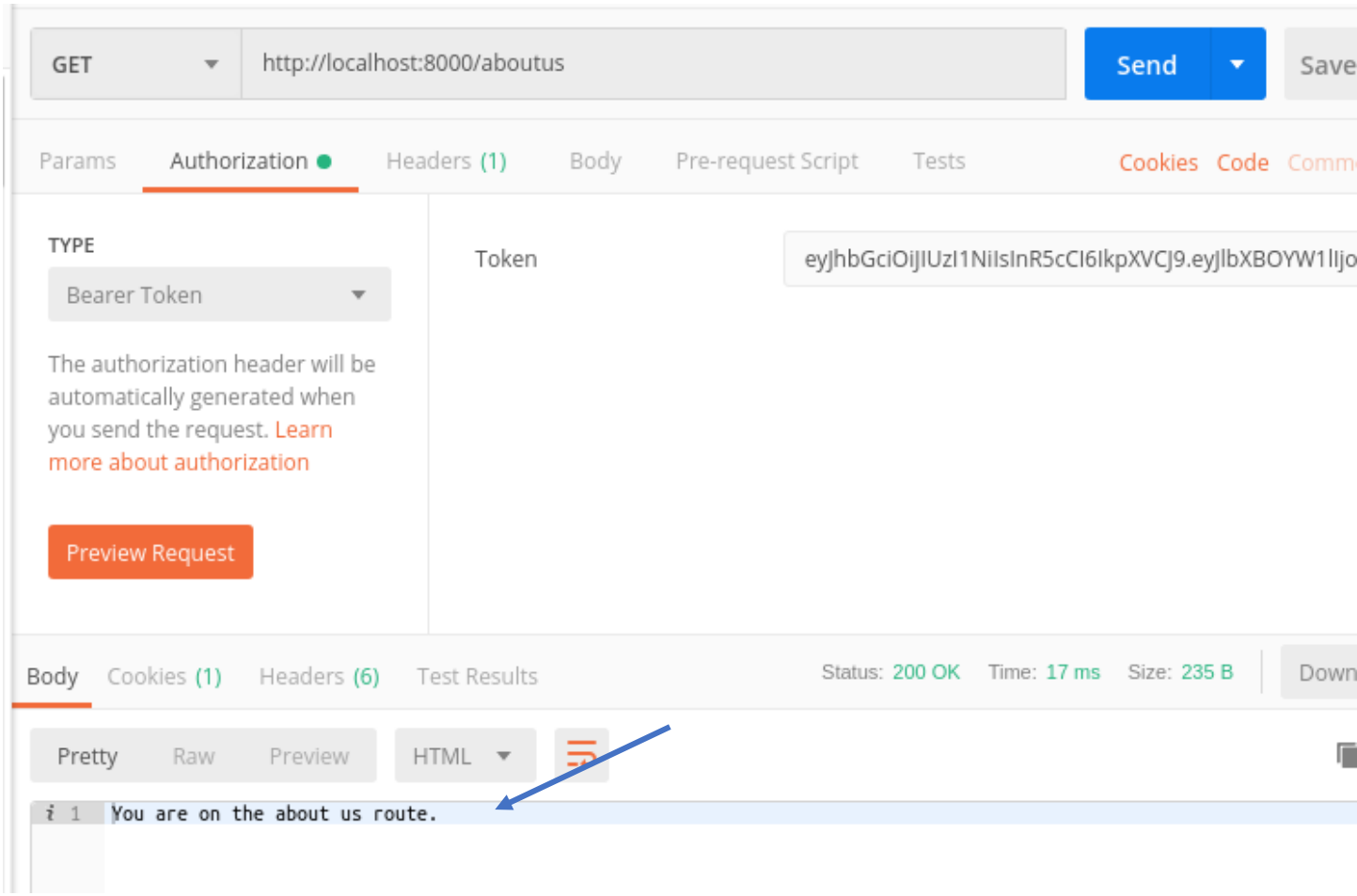
10. Now lets make the same request by passing in the token we generated on a previous tab



11. 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



12. You may try to manually change the token, for example remove the first "e" and hit send, the request will be denied