Table of Contents

Enhancing the site with 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.

PART 01 – INTRODUCTION TO MONGODB

Before proceeding, either open a new terminal window or tab. For working with MongoDB via the command line, you do not have to be in any particular directory within the terminal window.

Note: if you do not have MongoDB installed, install it now using:
`sudo apt install mongodb`
Assuming that you are on Ubuntu Linux 20+

In order to get into the MongoDB shell, use the command **sudo mongosh**
Note: for older versions use **sudo mongo** or **mongod**

1. Change the database to Weights and create a new table using the following code:

```
use Employees
```

2. Add a collection

```
db.createCollection("FTEmployees")
```

3. Perform a find(), it should not return anytihng but at least we know we now have a database and a collection

```
db.FTEmployees.find()
```

4. Enter a record

```
db.FTEmployees.insertOne( {empName : "Joe",  empPass : "1234" })
```

5. Verify the record.

```
db.FTEmployees.find()
```

6. Add another record by using the up arrow key and just changing the name and weight

```
db.FTEmployees.insertOne( {empName : "mary",  ",  empPass : "1234"})
```

7. Verify the new record

```
db. FTEmployees.find()
```

8. Lets change (update) Joe's record:

```
db.FTEmployees.update(
   {empName : "Joe"},
   {$set: {empPass : "Joe"} }
)
```

9. Verify the change

```
db.FTEmployees.find()
```

10. Enter a new document but this one will have a date in addition to the name and password

```
db.FTEmployees.insertOne(
{
empName : "Sally",
empPass : "1234",
Date : new Date()
}
)
```

11. Verify the change but this time chain the pretty() method

```
db.FTEmployees.find().pretty()
```

12. Finally update Joes's record to include a date and then do a find pretty

```
db.FTEmployees.update (
   {empName : "Joe"},
   {$set: {Date : new Date() } },
   false, false
)
```

------end of part 01-------

PART 02 – SETTING UP MONGOOSE

1. Return to the existing Node application and using a terminal pointing to your project, run the following install: **`npm install mongoose`**

   Mongoose is an ORM which interacts with the **`Employees`** database and abstracts away much of the annoyances of working directly with the database natively. Make sure you install this package in the project folder.

2. Create a new directory called **`models`** and touch a new .js file inside of models called employee.js and add the following lines. Do this using your editor which should have the application opened:

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/Employees', { useNewUrlParser: true });
```

   The first line is simply requiring the mongoose package and the second is using the **`connect()`** method which takes 2 parameters, the location of the **`mongod`** service and a json object which is required and standard according to the documentation.

3. Next we will define the schema with the name empSchema:

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/Employees, { useNewUrlParser: true } ');
const empSchema = new mongoose.Schema({
  empName: String,
  empPass: String,
  created: {type: Date, default: Date.now }
});
```

4. We also need to let the client files know which collection we are working with, so expand the code to include the collection name:

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/Employees, { useNewUrlParser: true });
const empSchema = new mongoose.Schema({
  empName: String,
  empPass: String,
  created: {type: Date, default: Date.now }
},{
    collection:'FTEmployees'
});
```

   Note, if you are using version 6+ of MongoDB you no longer need the object passed as the second parameter to the connect() method. It is greyed out here and in #6 below.

5. Finally for the employee.js file, we need to export our schema

```
module.exports = mongoose.model('Employees', empSchema);
```

6. Here is the entire file

```
const mongoose = require('mongoose');
mongoose.connect(
  'mongodb://localhost:27017/Employees,
  { useUnifiedTopology: true },
  { useNewUrlParser: true}
);
const empSchema = new mongoose.Schema({
  empName: String,
  empPass: String,
  created: {type: Date, default: Date.now }
},{
   collection:'FTEmployees'
});
//
module.exports = mongoose.model('Employees', empSchema);
```

At this point, test the application to make sure there are no errors.

------end of part 02-------

PART 03 – EXPANDING THE CONTROLLER FUNCTIONS TO WORK WITH DATABASE

1. Open controller.js in an editor and the first line will be a variable pointing to the **`models`** directory and its contents.

```
const Employee = require('../models/employee');
exports.getdefault=function(req, res){
     res.send('You are on the root route.');
};
//
```

2. Next we will change the **`getemployees`** function. That function will use the **`Employee`** variable created above and its attached **`find()`** method

```
exports.getemployees=function(req, res){
     Employee.find();
};
```

3. The find() method, like almost ALL Mongoose methods, takes an object as the first parameter and then, depending on the version, a function as the second. For a find all, the first parameter object must be blank.

```
exports.getemployees=function(req, res){
    Employee.find({})
    .then(
        employeeData => res.send(employeeData)
    )
};
```

Note: as of version 6+ we are now being forced to interact with the database asynchronously. This means that we either use async/await or chain a .then() method to the find() method.

4. In order to handle any errors we need to chain a .catch() method in addition to the then() method:

```
exports.getemployees=function(req, res){
    Employee.find({})
    .then(
        employeeData => res.send(employeeData)
    )
    .catch((err)=>{
        res.send(err);
    })
    //res.send('You are on the getdocs route.');
};
```

Now with this new code, we end the connection to the server if any errors occur and respond to the client with any data we got from executing the **`find()`** method.

5. In the routes.js file, make sure we have a route to match the function

```
router.get('/getemployees', controller.getemployees);
```

6. Test the code by opening a browser and navigating to **`http://localhost:8000/getemployees`**


```
▼ 0:
    _id:        "5f3d28d7694d1795e92d97ea"
    empName:    "Joe"
    empWeight:  96.5
    Date:       "2020-08-19T13:30:20.238Z"
    created:    "2020-08-19T14:44:18.734Z"
▼ 1:
    _id:        "5f3d2945694d1795e92d97eb"
    empName:    "Mary"
    empWeight:  65.7
    created:    "2020-08-19T14:44:18.735Z"
▼ 2:
    _id:        "5f3d295a694d1795e92d97ec"
    empName:    "Sally"
    empWeight:  65.9
    Date:       "2020-08-19T13:30:02.506Z"
    created:    "2020-08-19T14:44:18.735Z"
```

Here is the entire function

```
exports.getemployees=function(req, res){
            Employee.find({})
            .then(
               employeeData => res.send(employeeData)
            )
            .catch((err)=>{
               res.send(err);
            })
};
```

7. We can try to get just one employee. First in **`controller`** create a controller method called **`getemployee`**.

```
    exports.getemployee = function(req, res) {

    };
```

8. Notice, this method is implying singularity. We can now try to get a single record by passing in the *name* to get in the url .Add a route to the routes.js file

```
router.get('/getemployee/:employeeName', controller.getemployee);
```

Notice the colon and the path name after the path. This will accept any parameters passed by the user into the getemployee() function.

9. The **`getemployee()`** method will interrogate the **request** object like before, but this time we are looking into the **parameter** property (called params). In the code below we are asking the params property for the value in employeeName, that the user was supposed to pass to this route:

```
    exports.getemployee = function(req, res) {
       let empToFind = req.params.employeeName;

    };
```

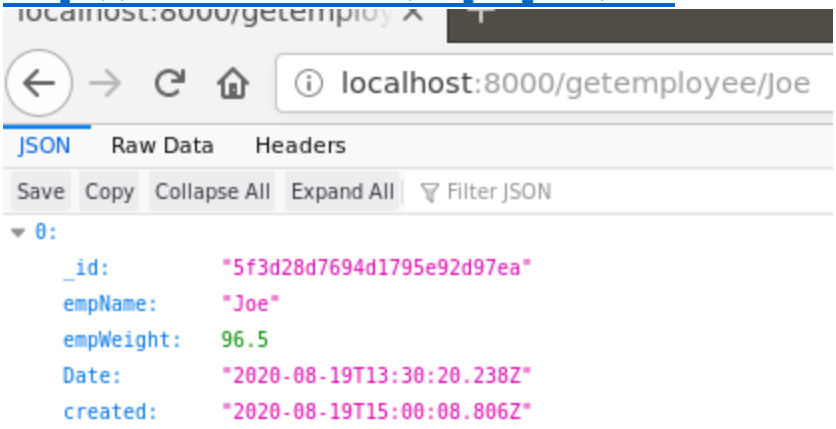
I could have used empName instead.

10. We can now pass this value to the find() method of our Employee object and handle any errors, as well as the result of our search:

```
       let empToFind = req.params.employeeName;
       Employee.find({empName:empToFind})
       .then(
          employeeData => res.send(employeeData)
       )
       .catch((err)=>{
          res.send(err);
       })
```

Note this is almost exactly the code for the getemployees() function, the only difference is that we passed an object to be searched.

11. Test the code by opening a browser and navigating to http://localhost:8000/employees/Joe



Of course you can test in Postman also

12. (optional) We can cater for no records found by adding a simple if statement. Here is the entire function

```
Employee.find({empName:empToFind})
.then(
   employeeData => {
      if(employeeData.length === 0)
         res.send("No data!");
      else
         res.send(employeeData);
   }
)
.catch((err)=>{
   res.send(err);
})
```

13. (Optional) The above will return an empty array if not configured. However we know it is already configured to work with JSON, so lets return JSON if no records found:

```
   employeeData => {
      if(employeeData.length === 0)
         res.send({"message":"No Data!"});
      else
         res.send(employeeData);
   }
```

The entire getemployee function

```
exports.getemployee= function(req, res){
        let empToFind = req.params.employeeName;
        Employee.find({empName:empToFind})
        .then(
            employeeData => {
                if(employeeData.length === 0)
                    res.send({"message":"No Data!"});
                else
                    res.send(employeeData);
            }
        )
        .catch((err)=>{
            res.send(err);
        })
};
```

The entire routes.js file so far:

```
const controller = require('./../controllers/controller');
      module.exports = function(router){
        router.get('/', controller.getdefault);
        router.get('/aboutus', controller.aboutus);
        router.post('/addemployee', controller.addemployee);
        router.get('/getemployees', controller.getemployees);
        router.get('/getemployee/:employeeName', controller.getemployee);
}
```

------end of part 03-------

1. In the routes.js file, you should already have a function called addemployee. If not copy any of the previous route lines and change the route to be add a new document.

```
router.post('/addemployee', controller.addemployee);
```

Notice that the method call is a **`post()`** NOT **`get()`**.

2. If you do not have a corresponding function, create a matching function in the controller.js file, in fact we can just copy, paste and edit the **`deletebyname()`** function.

```
exports.addemployee = function(req, res){
   let empName = req.body.empName;
   let empPass = req.body.empPass;
};
```

In this function, we get the name and new employee from an HTML form, NOT the **`URL`**.

3. Create a variable called Emp and point it to the **`Employee`** object, which represents our database. Remember in line 1 of the controller.js file we required the employee.js file:

```
exports.addnewdoc = function(req,res){
    let empName = req.body.empName;
    let empPass = req.body.empPass;
    const Emp = new Employee();
```

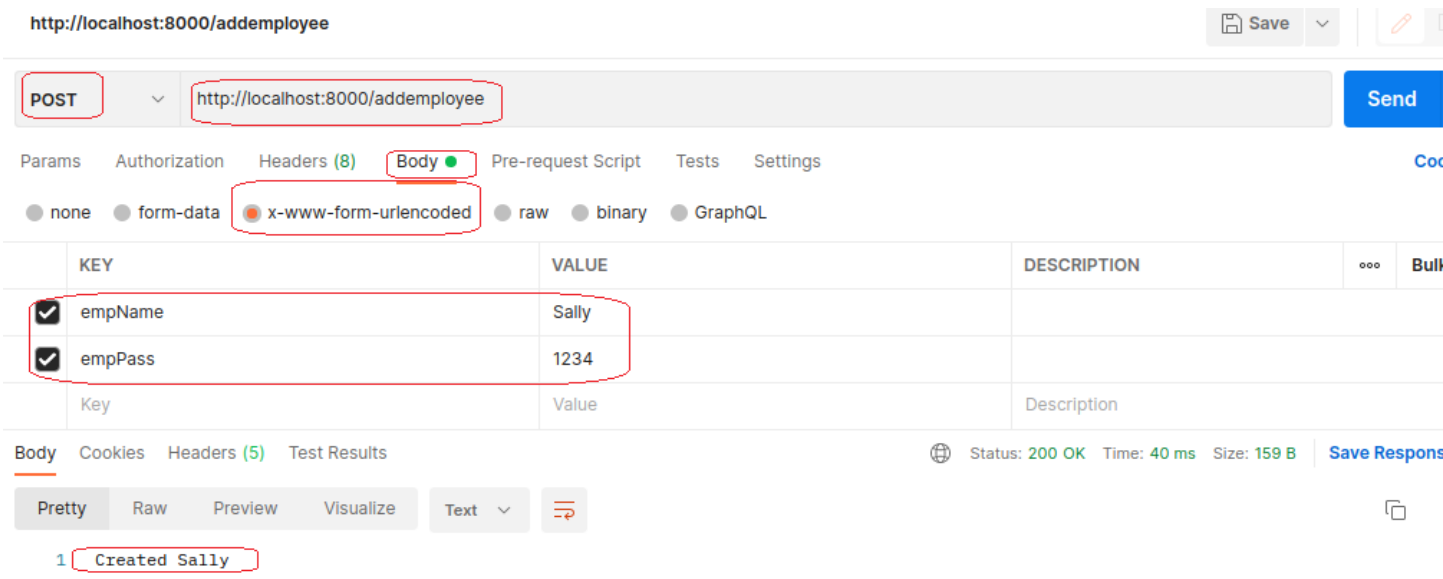
4. Use the new variable, Emp, and its properties to pass values from the form to the database properties

```
    const Weights = new Weight();
    Emp.empName = empName;
    Emp.empPass = empPass;
```

5. Now all we have to do is call the **`save`**() method of our **`Employee`** object and deal with errors, here is the entire function

```
    Emp.empName = empName;
      Emp.empPass = empPass;
    Emp.save()
    .then(msg => {
       res.send({"message":"Created " + Emp.empName});
    })
```

As usual in an asynchronous operation, you have to add the then() method, pass a parameter to accept any return from the save() method and then call the send() method to pass that value back to the user



You can also verify that Sally is in the database by going to localhost:8000/getemployees

6. As we did before, you should also chain the catch() method to handle any errors:

```
    Emp.empName = empName;
    Emp.empPass = empPass;
    Emp.save()
    .then(msg => {
       res.send({"message":"Created " + Emp.empName});
    })
    .catch(
       err => res.send({"message":err.message})
    );
```

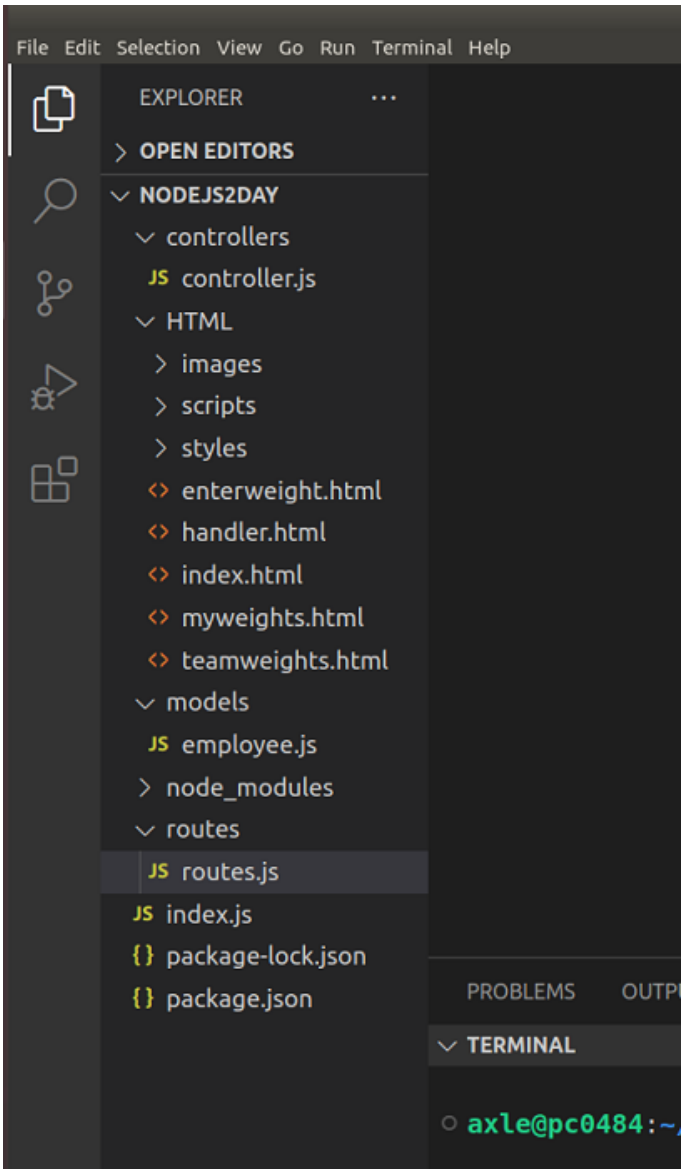
In this scenario there is a *message* property attached to the err object

7. The entire addemployee() function

```
exports.addemployee=function(req, res){
    let empName = req.body.empName;
    let empPass = req.body.empPass;
    const Emp = new Employee();
    Emp.empName = empName;
    Emp.empPass = empPass;
    Emp.save()
    .then(msg => {
       res.send({"message":"Created " + Emp.empName});
    })
    .catch(
       err => res.send({"message":err.message})
    );
};
```

------end of part 04-------

PART 05 – 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. (Not necessary anymore, but if you run into an issue, definitely turn it 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 API. Unzip that folder inside of the project folder you created on Day01. There should be two .html files and three folders. We won't be using all the files.

e. Initially, the starter files for day 2 folder will be named HTML.zip, unzip this file directly in the FSD folder.

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. Let's 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>.

   You can continue working with VS Code, all files and folders should now be in your parent folder fsd.

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>
```

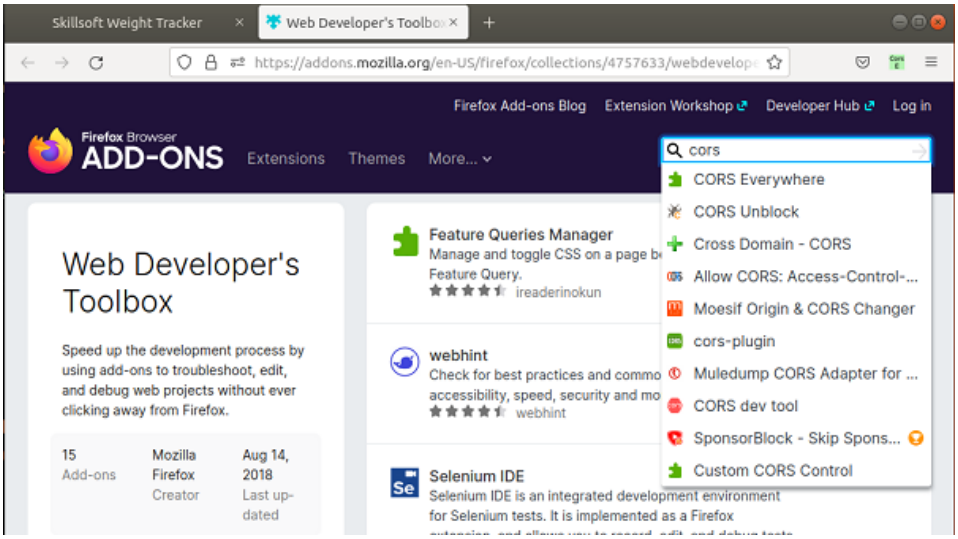
3. In the scripts.js file we can start writing the **`getData()`** function, put this code at the top of the document:

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

4. 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.

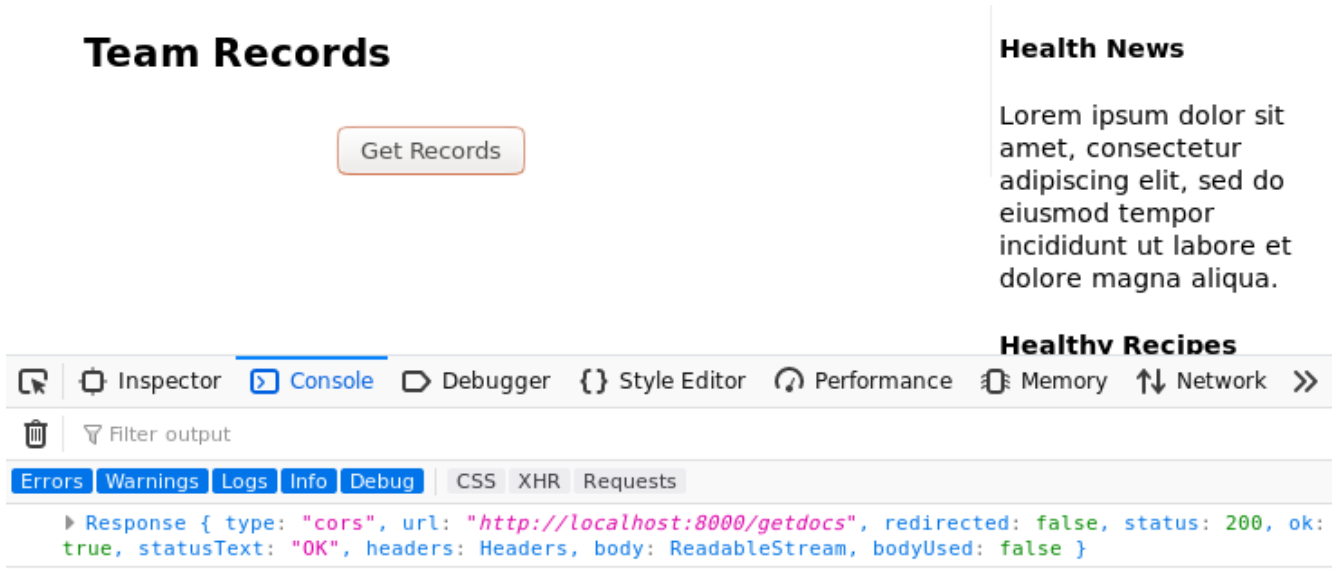
```
function getData(){
  fetch("http://localhost:8000/getemployees").then();
}
```

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 have to 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

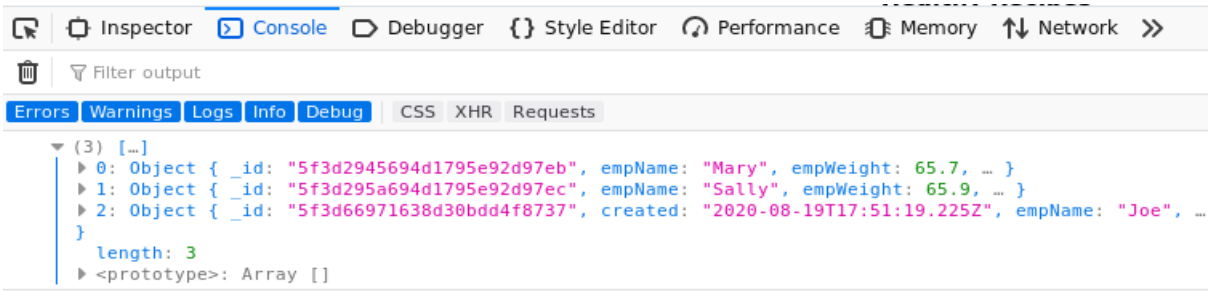
```
function getData(){
    fetch("http://localhost:8000/getemployees").then(function(response){
            return(response.json());
    });
}
```

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, 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);
    });
  })
};
```

PART 06 – DISPLAY THE DATA

From the part above, we now have an array that contains our data, all we need to do now is show this data. There is already a function called displayData() in the scripts.js file, so just pass the array to this function.

1. In the scripts.js, call **`displayData()`** and pass the data variable which contains the array:

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

2. Test by opening the allemployees.html file and clicking the Get Records button

3. If you want, you can convert the getData() function to use arrow function syntax instead of the traditional function keyword:

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

Here are the two functions so far using fetch()

```
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);
  }
}
```

PART 07 – USING ASYNC/AWAIT

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);
  }
};
```

PART 08 – ADDING A NEW PROFILE

We will use the HTML file given in the set of starter files. Look for the addemployee.html file and we will configure it to pass data from that form into the database, via the API endpoint. 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 in error and also allows us to control when the submit event occurs and how to control the data from the form.

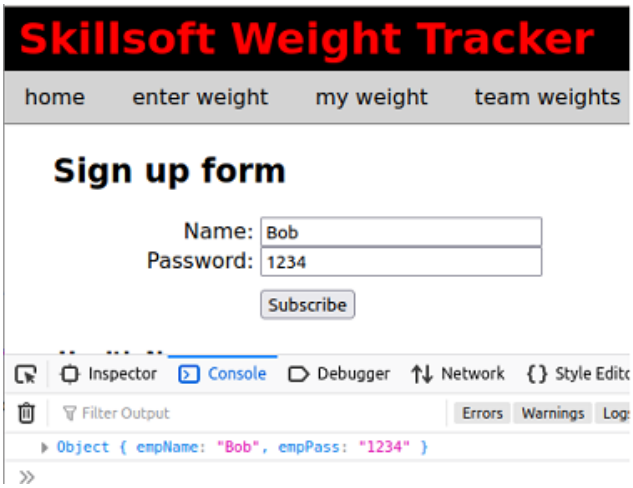
4. The next two lines will first get a handle to the form itself and then use the JavaScript **FormData()** method to extract the two fields into an object:

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

5. **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 into 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 from already wrapped up with field/value pairs.



Note: Object.formEntries is available by default via specification ECMAScript 2017

PART 09 – POSTING THE DATA

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 using the same fetch() method:

```
    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 itself is an object on the right side and we also wrap up the form object into JSON using the stringify() method of JSON.

4. At this point we have everything we need. Remember that the **fetch()** method returns a Promise object and unless you handle the Promise in the proper way, we will not get a message that the values were 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.

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);
        });
});
```

The bonus suction will be covered if time permits

BONUS SECTION – 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 })

};
```

   Handle methods asynchronously, so we need a then() method to start.

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:

```
            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 => {
    //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(
       {
        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: