Table of Contents

[**SECTION 1 – INTERRUPTING THE FORM SUBMIT FUNCTIONALITY** 2](#_Toc57975397)

[**SECTION 2 – ADDING MESSAGES TO THE USER** 3](#_Toc57975398)

[**SECTION 3 – ADDITIONAL VALIDATION with Regular Expressions** 4](#_Toc57975399)

[**Section 04 – CONNECTING TO THE APIS Using fetch()** 6](#_Toc57975400)

[**Section 05 – Display the Data** 10](#_Toc57975401)

[**Section 06 – using the modern async/await** 12](#_Toc57975402)

[**Appendix A – Using the Map Method to DIsplay Data** 12](#_Toc57975403)

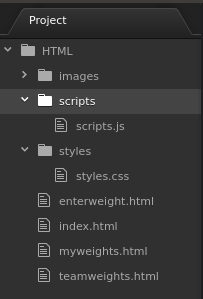
**Enhancing the sit with JavaScript**

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.

## SECTION 1 – INTERRUPTING THE FORM SUBMIT FUNCTIONALITY

1. Using any method you want, create a folder called scripts and an empty JavaScript file inside of that folder called scripts.js
2. We need to connect the file from #1 above to each of our HTML files, so between the ending </footer> tag and the ending </body> tag, insert this line:  
   <script src="scripts/scripts.js"></script>

|  |
| --- |
| **<footer>**  **<hr />**  **Copyright &copy; 2020. All rights reserved**  **</footer>**  **<script src="scripts/scripts.js"></script>**  **</body>**  **</html>** |



1. In scripts.js we could start writing a function to check each field on the form in order to determine if it is empty and if it is not empty, is the value in the proper format for our back end, so start this function in the js file:

|  |
| --- |
| **function validateForm(){**    **}** |

1. If you unzip the file for this section, inside is an HTML file called handler.html. we will be using this file just as a check that our form submission was complete. It does not mean the form values were submitted, this handle page is just to confirm that the form was submitted and that the code that handles that worked.
2. Give your form a name:

|  |
| --- |
| **<div id="container">**  **<main>**  **<h2>Enter your weight</h2>**  **<form name="frmCollectWeights" action="handler.html">**  **<div>**  **<label for="empName">Your Name</label>** |

1. We will interrupt the form submit from the **onsubmit()** function in the **form** tag, so add this *attribute* and *value* to the form tag (in enterweight.html):

|  |
| --- |
| **<div id="container">**  **<main>**  **<h2>Enter your weight</h2>**  **<form name="frmCollectWeights" action="handler.html" onsubmit = "return(validateForm());">**  **<div>**  **<label for="empName">Your Name</label>** |

Notice that the function name is the same in both files, the .html file and the .js file.

1. Insert a line into the **validateForm()** function on the scripts.js file to return false.

|  |
| --- |
| **function validateForm(){**  **return false;**  **}** |

Try to submit the form using just the browser, it should not go to the handler page. If you comment out the **return false** part, the form will be submitted.

1. Start checking the form fields for values, add this code inside the **validateForm()** function, also turn the last statement into a true:

|  |
| --- |
| **function validateForm(){**  **let empName = document.forms["frmCollectWeights"]["empName"];**  **let empWeight = document.forms["frmCollectWeights"]["empWeight"];**  **//**  **if (empName.value == "") {**  **return false;**  **}**  **if (empWeight.value == "") {**  **return false;**  **}**  **//**  **return true;**  **}** |

Try to submit the form by filling one or none or both of the fields, it is only when both fields are filled in with some value does the form complete the posting process.

## SECTION 2 – ADDING MESSAGES TO THE USER

1. We may want to inform the user when there is a problem submitting the form, so add a pair of **span** tags next to the input boxes in enterweight.html to accept the message from the script

|  |
| --- |
| **<div>**  **<label for="empName">Your Name</label>**  **<input id="empName" type="text" /><span id="nameMessage"></span>**  **</div>**  **<div class="formSeparator">**  **<label for="empWeight">Your Weight Today</label>**  **<input id="empWeight" type="text" /><span id="weightMessage"></span>**  **</div>** |

Now we can use the **innerHTML** of these span tags to pass messages from the script file to the user. Of course we can then style these tags based on our needs and preferences.

1. Add the lines to first pass a message to the offending field, and also put the focus back into the box that has the problem

|  |
| --- |
| **if (empName.value == "") {**  **document.getElementById("nameMessage").innerHTML="Name cannot be empty!";**  **empName.focus();**  **return false;**  **}**  **//**  **if (empWeight.value == "") {**  **document.getElementById("weightMessage").innerHTML="Weight cannot be empty!";**  **empWeight.focus();**  **return false;**  **}** |

1. If we test it now, we may notice that the message is not going away, so add these lines at the top of the function in the script file:

|  |
| --- |
| **function validateForm(){**  **document.getElementById("nameMessage").innerHTML="";**  **document.getElementById("weightMessage").innerHTML="";**  **let empName = document.forms["frmCollectWeights"]["empName"];**  **let empWeight = document.forms["frmCollectWeights"]["empWeight"];** |

this may not be enough but for now it will clear up at least one field until the form is posted

## SECTION 3 – ADDITIONAL VALIDATION with Regular Expressions

1. A name should not contain numbers, so lets add validation for that, however one of the easiest ways to check for numbers is to use regular expressions.

|  |
| --- |
| **let empName = document.forms["frmCollectWeights"]["empName"];**  **let empWeight = document.forms["frmCollectWeights"]["empWeight"];**  **let alphaOnly = /^[A-Za-z]+$/;** |

This variable defines a pattern

1. Awe will now use the match() method of JS and pass to it *alphaOnly*. By placing a ! in front of the entire statement, we are negating any match, so if we do not find only alphabetic characters, we reject the value:

|  |
| --- |
| **if (!empName.value.match(alphaOnly)) {**  **document.getElementById("nameMessage").innerHTML="Name cannot contain numbers!";**  **empName.focus();**  **return false;**  **}** |

1. (Optional) Lets also check the name for length, but we do it only when there is some value in the field entered by the user

|  |
| --- |
| **if (empName.value.length < 3 && empName.value != "") {**  **document.getElementById("nameMessage").innerHTML="Name too short!";**  **empName.focus();**  **return false;**  **}** |

1. We could also check the weight field for characters other than numbers

|  |
| --- |
| **if (isNaN(empWeight.value)) {**  **document.getElementById("weightMessage").innerHTML="Weight must be a number";**  **empWeight.focus();**  **return false;**  **}** |

Here is the entire scripts.js file so far:

|  |
| --- |
| **function validateForm(){**  **document.getElementById("nameMessage").innerHTML="";**  **document.getElementById("weightMessage").innerHTML="";**  **let empName = document.forms["frmCollectWeights"]["empName"];**  **let empWeight = document.forms["frmCollectWeights"]["empWeight"];**  **let alphaOnly = /^[A-Za-z]+$/;**  **//**  **if (empName.value == "") {**  **document.getElementById("nameMessage").innerHTML="Name cannot be empty!";**  **empName.focus();**  **return false;**  **}**  **if (!empName.value.match(alphaOnly)) {**  **document.getElementById("nameMessage").innerHTML="Name cannot contain numbers!";**  **empName.focus();**  **return false;**  **}**  **if (empName.value.length < 3 && empName.value != "") {**  **document.getElementById("nameMessage").innerHTML="Name too short!";**  **empName.focus();**  **return false;**  **}**  **if (empWeight.value == "") {**  **document.getElementById("weightMessage").innerHTML="Weight cannot be empty!";**  **empWeight.focus();**  **return false;**  **}**  **if (isNaN(empWeight.value)) {**  **document.getElementById("weightMessage").innerHTML="Weight must be a number";**  **empWeight.focus();**  **return false;**  **}**  **//**  **return true;**  **}** |

## Section 04 – CONNECTING TO THE APIS Using fetch()

Note:

1. Your API must be running in order for your code in this section to work. If it is not running, go to the folder Day01 and run the nodemon command or npm start.
2. Also make sure your CORS plugin on the browser is turned on.
3. 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
4. We will be using mainly the teamweights.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 just like we did for enterweight.html (if it hasn’t been already).
5. From the **main** div, remove all the dummy text and just include a **div** to display the data from our file, and a button to call a function to get the data

|  |
| --- |
| **<div id="container">**  **<main>**  **<h2>Team Records</h2>**  **<div id="records"></div>**  **<button onclick="getData();">Get Records</button>**  **</main>** |

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

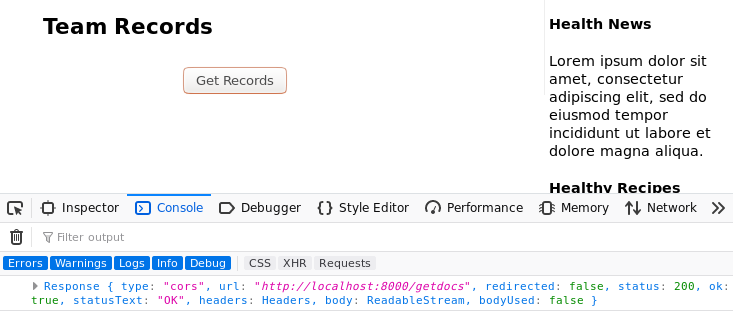
|  |
| --- |
| **function getData(){**  **fetch("http://localhost:8000/getdocs");**  **}** |

1. 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/getdocs").then();**  **}** |

1. 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:

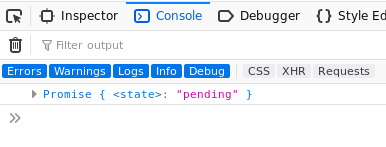
|  |
| --- |
| **function getData(){**  **fetch("http://localhost:8000/getdocs").then(function(response){**  **console.log(response);**  **});**  **}** |



This is a lot of text to filter through.

1. 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.

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

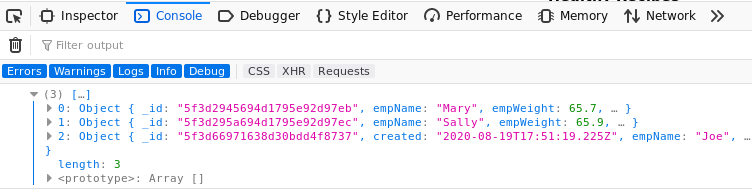
|  |
| --- |
| **function getData(){**  **fetch("http://localhost:8000/getdocs").then(function(response){**  **return(response.json());**  **});**  **}** |

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

|  |
| --- |
| **function getData(){**  **fetch("http://localhost:8000/getdocs").then(function(response){**  **return response.json()).then();**  **});**  **}** |

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

|  |
| --- |
| **function getData(){**  **fetch("http://localhost:8000/getdocs").then(function(response){**  **return(response.json()).then(function(data){**  **console.log(data);**  **});**  **})**  **}** |

  
Finally, we have the data we were looking for.

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

|  |
| --- |
| **function getData(){**  **fetch("http://localhost:8000/getdocs")**  **.then(function(response){**  **return(response.json())**  **.then(function(data){**  **console.log(data);**  **});**  **})**  **};** |

1. 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/getdocs")**  **.then(function(response){**  **return(response.json())**  **.then(function(data){**  **console.log(data);**  **}).catch(function(err){**  **console.log(err);**  **});**  **})**  **};** |

1. Using arrow functions

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

## Section 05 – Display the Data

1. Remember we had a **div** tag in the teamweights.html file that we can use to display the data, this **div** has an **id** of **records**. 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("records").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.

1. 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 + " weighed " + arr[i].empWeight + " Kgs</p>";**  **}**  **document.getElementById("records").innerHTML = outHTML;**  **}** |

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

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

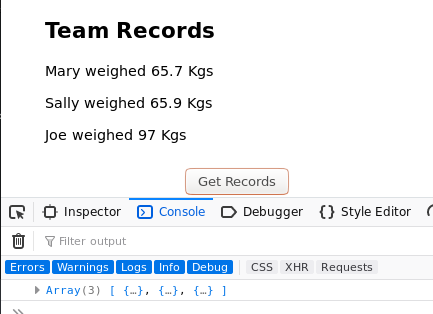
1. What if you also wanted to log the data. As it turns out we could have multiple **then()** methods in a structure. But because we already used the **response** within a function, we would have to manually pass it to the next chained event by using a **return** statement. In this case we would need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

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

1. 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/getdocs")**  **.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



## Section 06 – using the modern async/await

In order to use the **async/await** method, 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/getdocs");**  **const data = await response.json();**  **displayData(data);**  **};** |

With error handling:

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

## 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;**  **}** |