# Full Stack Web Developer

## Table of Contents

How to read this document ........ 3
Day01 HTML Structure ........ 4
PART 1 – BUILDING THE HTML STRUCTURE ........ 4
PART 2 – ADDING CSS TO THE HTML ........ 6
PART 3 – ADDING AN ASIDE & SECTIONS TO THE HTML ........ 9
PART 4 – ADDING STYLES TO THE ASIDE ........ 12
PART 5 – ADDING SEMANTICS ........ 15
PART 6 – USING TEMPLATES ........ 18
PART 7 – ADDING STYLES TO THE FORM ........ 19
Day02 JavaScript ........ 23
PART 1 – GETTING STARTED WITH JS ........ 23
PART2 – TRANSFER TO EXTERNAL JS ........ 26
PART 3 – ADDING A DUMMY LANDING PAGE ........ 27
PART 4 – INTERRUPTING THE FORM SUBMIT FUNCTIONALITY ........ 29
PART 6 – ADDING ADDITIONAL VALIDATION ........ 31
PART 7 – CONNECTING TO THE APIS - THE OLD WAY ........ 33
PART 8 – THE NEW WAY ........ 35
PART 9 – DISPLAY THE DATA ........ 35
PART 10 – USING THE APIS ........ 36
PART 11 – USING THE ULTRA MODERN ASYNC/AWAIT ........ 37
Day03 jQuery and AngularJS ........ 38
PART 1 – CHANGING OVER FROM JS TO JQUERY ........ 38
PART 2 – JQUERY VALIDATION ........ 39
PART 3 – JQUERY API FUNCTIONALITY ........ 42
PART 4 – CHANGING OVER FROM JS TO ANGULARJS ........ 44
PART 5 – ANGULAR'S HTTP SERVICE ........ 47
PART 6 – USING ANGULAR'S NG-REPEAT DIRECTIVE ........ 49
PART 7 – POSTING NEW DOCUMENTS WITH ANGULARJS ........ 50

Day04 Installing Node, MySQL & MongoDB ........ 53

**PART 1 – INSTALLING NODEJS, MYSQL AND MONGODB** ........ 53

**PART 2 – CONFIGURING A NEW USER IN MYSQL** ........ 57

**PART 3 – MYSQL DATABASES AND TABLES** ........ 58

**PART 4 – MYSQL AND NODEJS** ........ 59

**PART 5 – MYSQL AND NODEJS API** ........ 61

**PART 6 – CREATING MONGODB DATABASES AND COLLECTIONS** ........ 63

**PART 7 – INTEGRATING WITH NODEJS** ........ 65

Day05 Building the APIs ........ 72

**PART 1 – VERIFY NODEJS** ........ 72

**PART 2 – BUILDING A SIMPLE NODEJS APP** ........ 73

**PART 3 – INCLUDING AND WORKING WITH NODE PACKAGES** ........ 75

**PART 4 – ROUTING BASICS** ........ 77

**PART 5 – DECOMPOSING ROUTES** ........ 79

**PART 6 – DECOMPOSING CONTROLLERS** ........ 81

**PART 7 – INTEGRATING AND WORKING WITH MONGODB** ........ 83

**PART 8 – CRUD OPERATIONS WITH MONGODB** ........ 84

**PART 9 – SETTING UP MONGOOSE** ........ 86

**PART 10 – EXPANDING THE CONTROLLER FUNCTIONS TO WORK WITH DATABASE** ........ 88

**PART 11 – EXPANDING THE CONTROLLER TO DELETE FROM DATABASE** ........ 89

**PART 12 – EXPANDING THE CONTROLLER TO ADD A NEW DOCUMENT TO THE DATABASE** ........ 91

**PART 13 – EXPANDING THE CONTROLLER TO UPDATE A DOCUMENT IN THE DATABASE** ........ 93

Day06 Deployment ........ 95

**PART 1 – FOLDER SETUP** ........ 95

**PART 2 – CONFIGURING TEAM WEIGHTS** ........ 97

**PART 3 – CONFIGURING MYWEIGHTS** ........ 98

**PART 4 – ADDING A NEW RECORD** ........ 101

**PART 5 – DELETING A RECORD (ADMIN FOLDER)** ........ 103

**PART 6 – EDITING A RECORD (ADMIN FOLDER)** ........ 106

Appendix ........ 111

**APPENDIX A - CORS** ........ 111

**APPENDIX B – EDITORS** ........ 112

**APPENDIX C – REST CLIENT** ........ 113

# How to read this document

1. Code builds on code, so the further you are along the document, the larger the file gets. The starter code is displayed in normal font, but if code is added to that original code, the new code is highlighted in yellow.

2. Technology words, commands and instructions are always given in a different font, `like this`.

3. Any kind of display, either on the web page itself or as a message to the final user of the application is shown in double quotations "like this".

4. The *pageheader* needs to inform the user of the content of the site, so add a pair of `h1` tags and the page header text such as: "Skillsoft Weight Tracker".

1. Actual code, for example, HTML, CSS and JavaScript are always shown inside of a box like this:

```
    Code
    Code
    code
```

5. Whenever we refer to one of the actual files for example index.html, the name of the file as it appears in the computer's file system is always underlined.

6. There are several **notes** along embedded in the document and these are bolded.

# Day01 HTML Structure

## Part 1 – Building the HTML structure

1. Add a pair of **`title`** tags between the **`head`** tags, then insert some content e.g. "Skillsoft Weight Tracker"

2. Add four pairs of **`div`** tags between the **`body`** tags in order to match the four horizontal parts of our final web page. Indent the four **`div`** tags.

3. Give an **`id`** name to each of the four **`div`** tags to correspond to the four major parts such as *pageheader*, *navigation*, *maincontent* and *footer* in that order.

4. The *pageheader* needs to inform the user of the content of the site, so add a pair of **`h1`** tags and the page header text such as: "Skillsoft Weight Tracker".

5. The navigation will be constructed with **`ul-li`** tags, so add one pair of **`ul`** tags and 5 pairs of **`li`** tags inside of the **`ul`** tags to represent 5 menu links.

6. Between the **`li`** tags, add the 5 internal links *home, enter weight, my weight, team weights* and *winner/loser.*

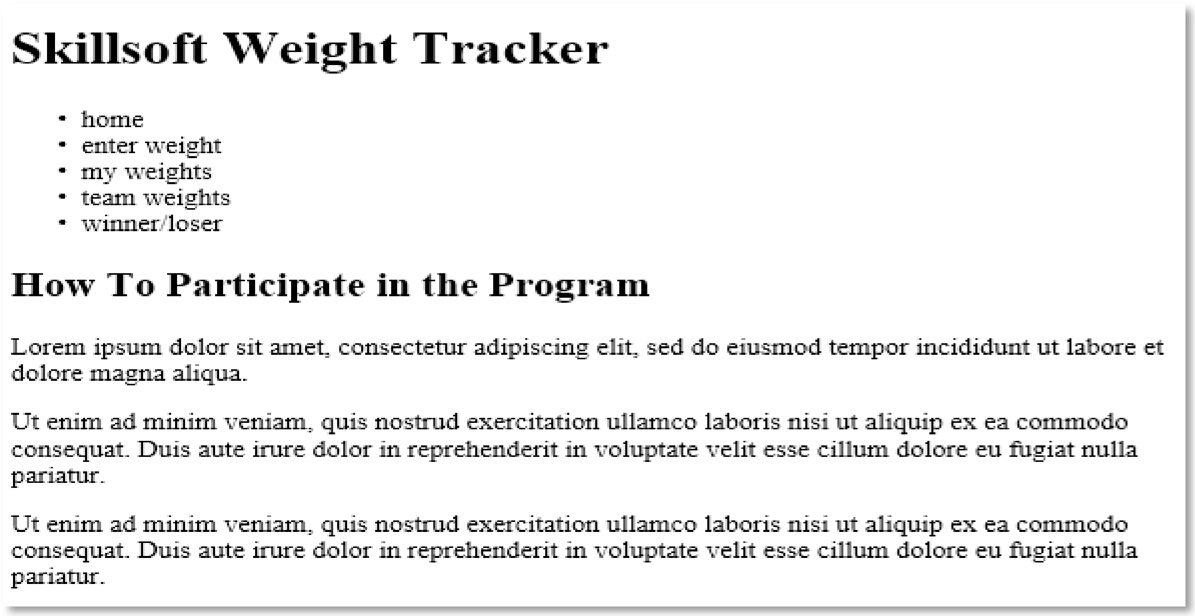
7. For the maincontent area we will need a sub-heading and some dummy content, so add a pair of **`h2`** tags and three **`p`** tags to hold lorem ipsum text. Enter the text of "How To Participate in the Program"

8. Add some *lorem ipsum* text between the three **`p`** tags.

9. Up to this point, the entire index.html file should look like what is shown below:

```
    <!DOCTYPE HTML>
    <html>
        <head>
                <title>Skillsoft Weight Tracker</title>
        </head>
        <body>
                <div id="pageheader">
                        <h1>Skillsoft Weight Tracker</h1>
                </div>
                <div id="navigation">
                        <ul>
                                <li>home</li>
                                <li>enter weight</li>
                                <li>my weights</li>
                                <li>team weights</li>
                                <li>winner/loser</li>
                        </ul>
                </div>
                <div id="maincontent">
                        <h2>How To Participate in the Program</h2>
                                <p>Lorem ipsum…
                                </p>
                                <p> Lorem ipsum…
                                </p>
                                <p> Lorem ipsum…
                                </p>
                </div>
                <div id="footer">
                </div>
        </body>
    </html>
```

The file when viewed in the browser should now look like the image below:



Skillsoft Weight Tracker

- home
- enter weight
- my weights
- team weights
- winner/loser

How To Participate in the Program

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

10. For the **footer div** add a horizontal rule and the copyright text: "Copyright 2019. All rights reserved"

11. We need to make the **li** items links as well as we need the page header to always point to index.html, so between the **h1** tags, under the **pageheader div**, add the **anchor** tags

```
        <body>
                <div id="pageheader">
                        <h1><a href="index.html">Skillsoft Weight Tracker</a></h1>
                </div>
                <div id="navigation">
```

12. Add **anchor** tags between the **li** tags and have them point to non-existent files for right now

```
                <div id="navigation">
                        <ul>
                                <li> <a href="index.html">home</a></li>
                                <li> <a href="enterweight.html">enter weight</a></li>
                                <li> <a href="myweights.html">my weights</a></li>
                                <li> <a href="teamweights.html">team weights</a></li>
                                <li> <a href="winnerloser.html">winner/loser</a></li>
                        </ul>
                </div>
```

## PART 2 – ADDING CSS TO THE HTML

1. In the styles folder there is a styles.css file with some styles in it already, open it in an editor and we can start styling the **anchor tag**, give it the style below:

```
    a {
        text-decoration:none;
    }
```

2. We would need to give our **pageheader div** a background color of black

```
    #pageheader{
        background-color:#000;
}
```

3. Lets style the **h1** tag inside of the **pageheader div** so that it has some space around it and it jams up against the top left corner. In this case **h1** is a decendent of the **pageheader div**.

```
    #pageheader h1 {
        padding: 5px 10px;
        margin:0;
    }
```

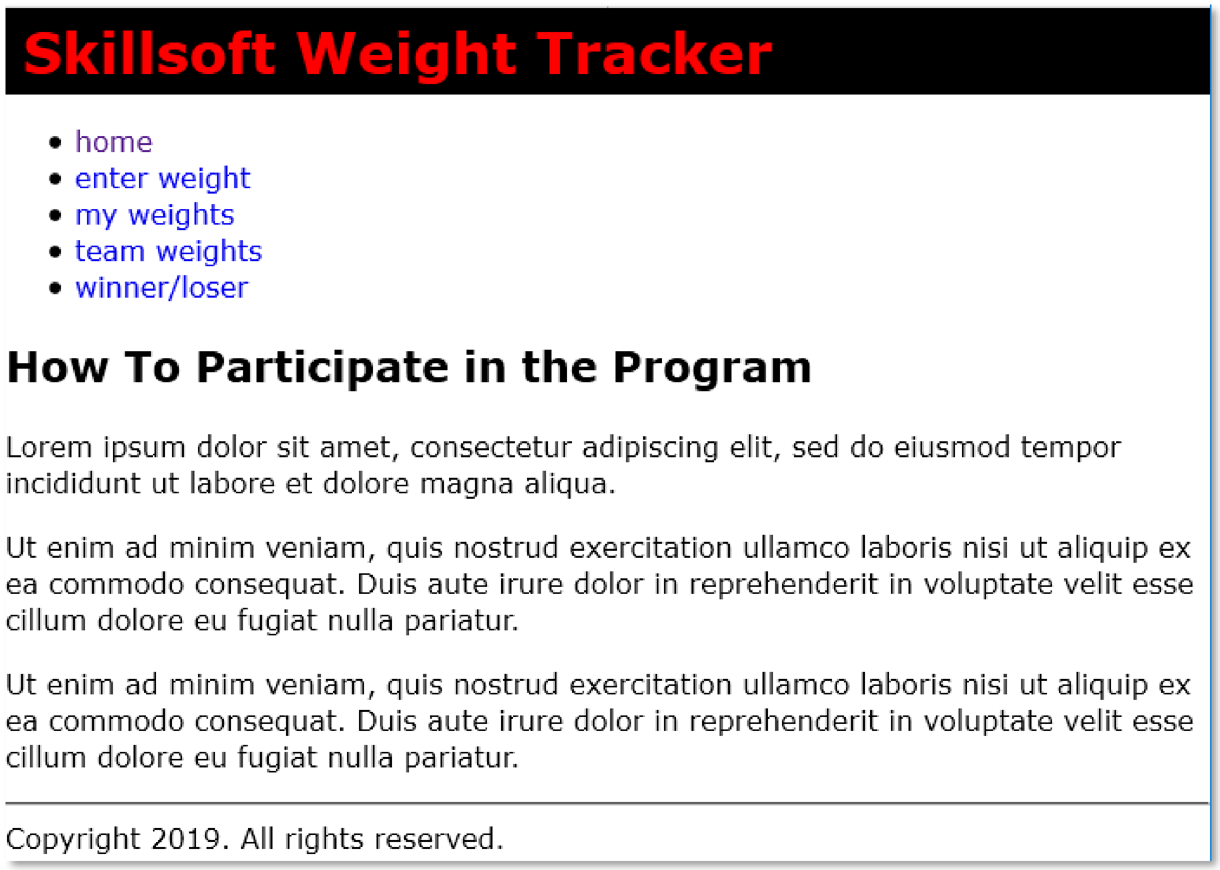
4. Turn the color of the **pageheader** text to red using the following style. Note we could not put the color into the style above because then the red text will have to compete with the black.

```
        #pageheader h1 a {
            color:red;
        }
```

5. Before going to far, connect our styles to our html, so in the **`head tags`** of the html document, add the following line to add our styles so far:

```
        <link rel="stylesheet" type="text/css" href="styles/styles.css" />
```

The rendered file at this point should look like the image below:



Skillsoft Weight Tracker

- home
- enter weight
- my weights
- team weights
- winner/loser

How To Participate in the Program

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Copyright 2019. All rights reserved.

6. Lets now work on the **`navigation div`**, add the following styles to the tag in a general way:

```
        #navigation{
            text-align:left;
            background-color:lightgray;
            border-bottom:1px solid gray;
        }
```

This would add a background color, align all text to the left and create a subtle line at the bottom of that structure.

7. Display both the list items inside of navigation and the **`anchor tags`** inside of navigation as inline block items. Use of a comma here prevents us from writing two blocks of code.

```
    #navigation li, #navigation a {
        display:inline-block;
    }
```

The style would remove the bullets and cause the navigation links to show up horizontally.

8. With the use of pseudo-classes we can change the state of an element, so lets access the hover state of the `anchor tags` inside of the navigation area and change the background colour to light yellow when the mouse hovers over the links.

```
    #navigation a:hover{
        background-color:lightyellow;
    }
```

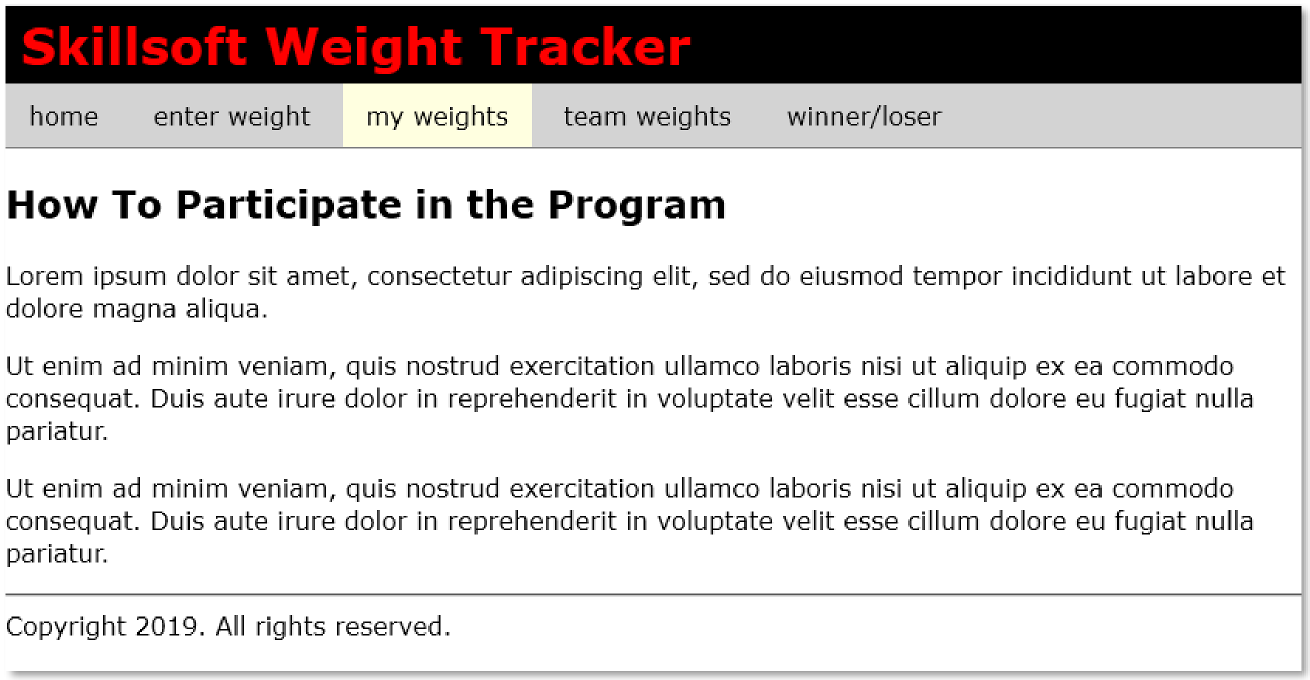
9. Lets change the navigation links to black instead of the default blue colour. Also add some space around each link so they don't look crammed.

```
    #navigation a{
        color:black;
        padding: 10px 15px;
    }
```

10. Remove any kind of spacing around the entire `ul` structure so that the navigation area is positioned directly next to the page header and no spaces are shown

```
    ul{
        padding:0;
        margin:0;
    }
```

11. The rendered html file should now look like the one in the image below:



How To Participate in the Program

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Copyright 2019. All rights reserved.

## PART 3 – ADDING AN ASIDE & SECTIONS TO THE HTML

1. Since the **aside** is inside of the middle part of the page, so basically the **maincontent** area, we need to insert the tags in between the **maincontent** pair of **div** tags, so just after the last **p** tag. They are **divs** now but they will become semantic soon.

```
    <p>Ut enim ad...
    </p>
      <div id="aside">
      </div>
    </div>
    <div id="footer">
```

2. Inside of this pair of aside tags are two parts that will eventually become sections, but for now, use **div** tags and give them ids of *section*.

```
    </p>
      <div id="aside">
        <div id="section">
        </div>
        <div id="section">
        </div>
      </div>
    </div>
    <div id="footer">
```

3. Add content to the sections, the first section will be for news so just give it a title by using an **h4** tag and some *lorem ipsum* text below inside of a pair of **p** tags.

```
<div id="section">
  <h4>Health News</h4>
  <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore
et dolore magna aliqua.
  </p>
</div>
```

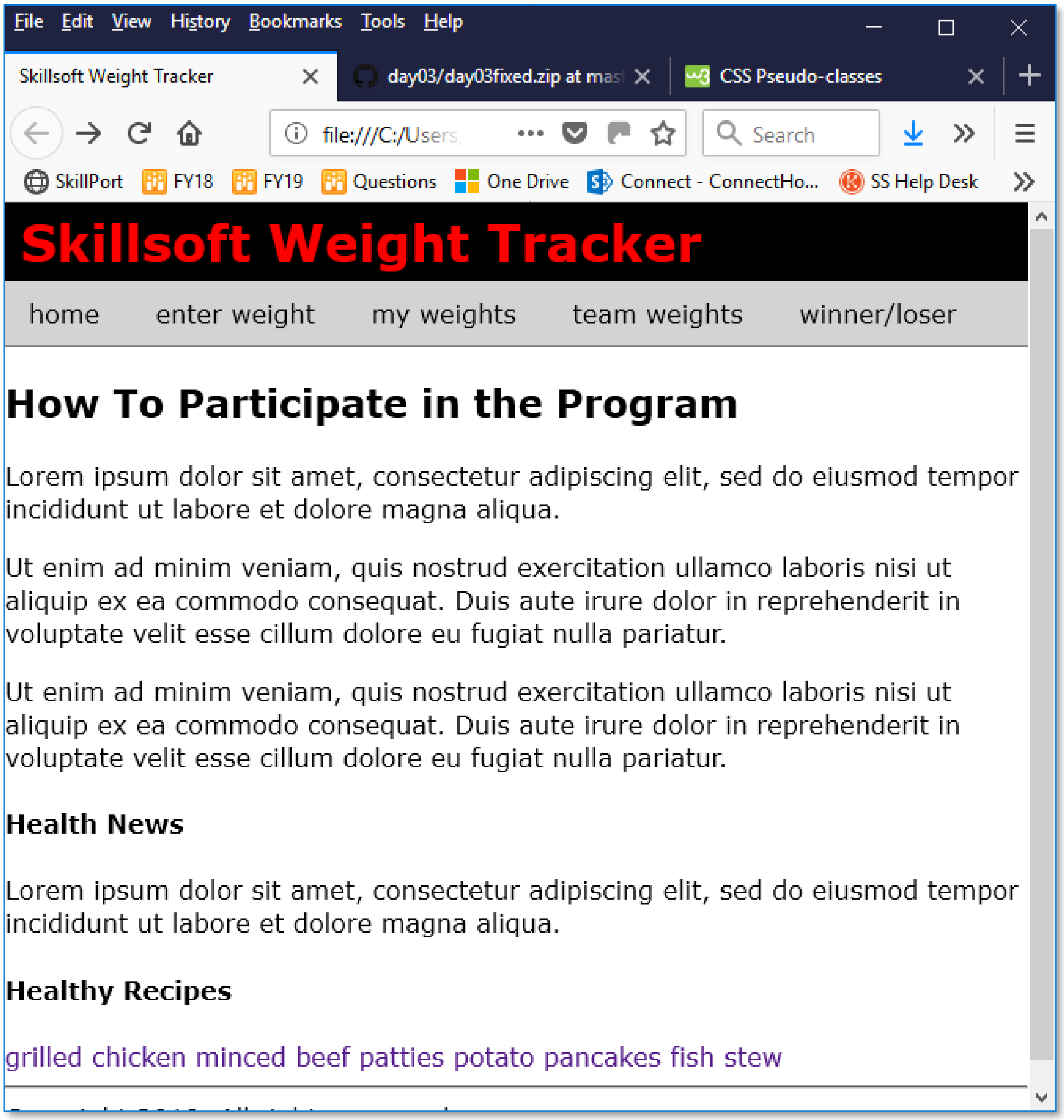
4. The bottom section will hold links to recipes, and it will also have an **h4** header, so enter the code below

```
<div id="section">
  <h4>Healthy Recipes</h4>
  <a href="">grilled chicken</a>
  <a href="">minced beef patties</a>
  <a href="">potato pancakes</a>
  <a href="">fish stew</a>
</div>
```

5. The entire *maincontent* area should now look like this

```
<div id="maincontent">
     <h2>How To Participate in the Program</h2>
     <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua.
     </p>
     <p>Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat
nulla pariatur.
     </p>
     <p>Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat
nulla pariatur.
     </p>
     <div id="aside">
               <div id="section">
                         <h4>Health News</h4>
                         <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua.
                         </p>
               </div>
               <div id="section">
                         <h4>Healthy Recipes</h4>
                         <a href="">grilled chicken</a>
                         <a href="">minced beef patties</a>
                         <a href="">potato pancakes</a>
                         <a href="">fish stew</a>
               </div>
     </div>
</div>
```

6. The rendered version should like the image below:


File Edit View History Bookmarks Tools Help
Skillsoft Weight Tracker
day03/day03fixed.zip at mas
CSS Pseudo-classes
file:///C:/Users
Search
SkillPort FY18 FY19 Questions One Drive Connect - ConnectHo... SS Help Desk

Skillsoft Weight Tracker

home enter weight my weights team weights winner/loser

How To Participate in the Program

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Health News

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Healthy Recipes

grilled chicken minced beef patties potato pancakes fish stew

Notice that it all looks like one big page, we will use CSS to style the future `asides` and `sections` shortly

## PART 4 – ADDING STYLES TO THE ASIDE

1. In order to have the **`aside`** and **`sections`** move to the right side of the page and leave the main content on the left we need to force sizes onto these big areas. Add the following style to the **`maincontent`** div in style.css:

```
#maincontent{
    float:left;
    width:560px;
    border-right: 1px solid #eeeeee;
}
```

2. At the same time add this style to the **`div`** with **`id`** of aside:

```
#aside{
    float:left;
    width:200px;
    padding-bottom:10px;
    padding-left: 4px;
}
```

3. If you render the file now it will show both areas and the footer is now up in the right top part of the page, we need to add a style to force that footer to stay at the bottom of the page:

```
#footer{
    clear:both;
    text-align:center;
    font-size:.8em;
    padding-top:50px;
}
```

4. The HTML page probably still does not move the **`aside`** and **`sections`** to the right of the page, this is because one is inside of the other, we need to make them equal parts of some whole, so we need a **container div**.

   Add a container **`div`** above the **`maincontent`** div and shift **`maincontent`** to the right and make it the same level as the **`aside`** div. Remember to close the **`maincontent`** div just before the **`aside`** div.

Do this in the index.html file.

```
<div id="maincontainer">
   <div id="maincontent">
         <h2>How To Participate in the Program</h2>
         <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
         </p>
         <p>Ut enim a...
         </p>
         <p>Ut enim ad ...
         </p>
   </div>
   <div id="aside">
         <div id="section">
               <h4>Health News</h4>
               <p>Lorem ipsum d....
               </p>
         </div>
         <div id="section">
               <h4>Healthy Recipes</h4>
               <a href="">grilled chicken</a>
               <a href="">minced beef patties</a>
               <a href="">potato pancakes</a>
               <a href="">fish stew</a>
         </div>
   </div>
</div>
```

5. Eventually we will wrap the entire page into a container so that we can center it, but for now the **maincontainer** and the **footer** divs need to have their **widths** contained a bit in preparation for that container, so add this style in style.css to both tags:

```
main, footer{
     width:90%;
     margin: 0 auto;
     margin-bottom: 60px;
}
```

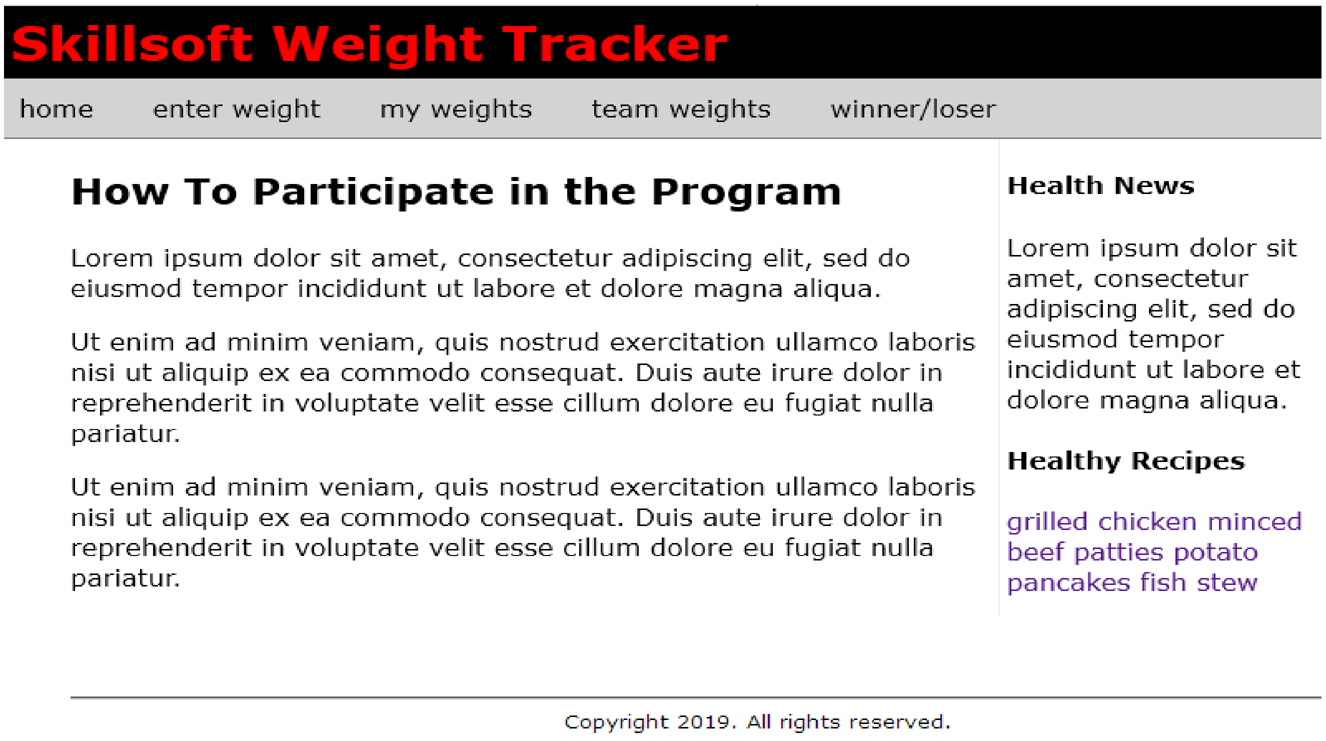
6. The links in the bottom section of the **aside** section should show up as individual links so add the following style. This will also add a subtle line between each link.

```
#aside #section a{
     display:block;
     padding: 6px;
     border-bottom: 1px solid lightgray;
     color:black;
}
```

7. The css file, from **`#maincontent`** should now look like this:

```
#maincontent{
    float:left;
    width:560px;
    border-right: 1px solid #eeeeee;
}
#aside{
    float:left;
    width:200px;
    padding-bottom:10px;
    padding-left: 4px;
}
#footer{
    clear:both;
    text-align:center;
    font-size:.8em;
    padding-top:50px;
}
#maincontainer, #footer{
    width:90%;
    margin: 0 auto;
    margin-bottom: 60px;
}
#aside #section a{
    display:block;
    padding: 6px;
    border-bottom: 1px solid lightgray;
    color:black;
}
```

The rendered HTML file should look like the image below:


Skillsoft Weight Tracker

home enter weight my weights team weights winner/loser

How To Participate in the Program

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Health News

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Healthy Recipes

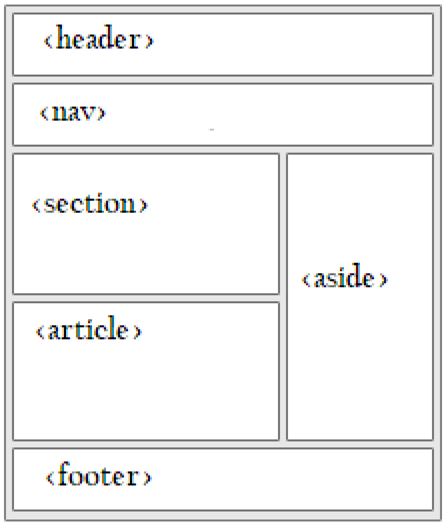
grilled chicken minced beef patties potato pancakes fish stew

Copyright 2019. All rights reserved.

**NOTE: you may need to stretch the browser window to see this view otherwise the aside will drop to the bottom.**

## PART 5 – ADDING SEMANTICS

Semantic tags help other machines read and interpret HTML documents. There are several of them, but here are the main ones in a diagram:



1. Visually inspect the rendered document in order to determine the various areas that apply to the semantic tags above:

```
<html>
    <head>
    </head>
    <body>
            <div class="wrapper">
                    <img id="logo" src="images/chart.gif" />
                    <div id=”pageheader”>
                            <h1>
                                    <a href="index.html">Skillsoft Weight Tracker</a>
                            </h1>
                    </div>
                    <div id=”navigation”>
                            <ul>
                                    <li><a href="index.html">home</a></li>
                            </ul>
                    </div>
                    <div id="maincontainer">
                            <div id="maincontent">
                                    <h2>How To Participate in the Program</h2>
                                    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
                                    </p>
                            </div>
                            <div id="aside">
                                    <div id="section">
                                            <h4>Health News</h4>
                                    </div>
                                    <div id="section">
                                            <h4>Healthy Recipes</h4>
                                            <a href="">grilled chicken</a>
                                    </div>
                            </div>
                    </div>
                    <div id="footer">
                            <hr />
                            Copyright 2019. All rights reserved.
                    </div>
            </div><
    </body>
</html>
```

2. The highlighted areas can now be converted to semantic tags

```
<html>
    <head>
            <title>Skillsoft Weight Tracker</title>
            <meta charset="utf-8"/>
            <link rel="stylesheet" type="text/css" href="styles/styles.css" />
    </head>
    <body>
            <div class="wrapper">
                    <img id="logo" src="images/chart.gif" />
                    <header>
                            <h1>
                                    <a href="index.html">Skillsoft Weight Tracker</a>
                            </h1>
                    </header>
                    <nav>
                            <ul>
                                    <li><a href="index.html">home</a></li>...
                            </ul>
                    </nav>
                    <main>
                            <div id="maincontent">
                                    <h2>How To Participate in the Program</h2>
                                    <p>Lorem ipsum dolor sit amet, ...
                            </div>
                            <aside>
                                    <section>
                                            <h4>Health News</h4>
                                            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit,
sed do eiusmod tempor incididunt ut labore et dolore...
                                            </p>
                                    </section>
                                    <section>
                                            <h4>Healthy Recipes</h4>
                                            <a href="">grilled chicken</a>...
                                    </section>
                            </aside>
                    </main>
                    <footer>
                            <hr />
                            Copyright 2019. All rights reserved.
                    </footer>
            </div><!--closes the container div-->
    </body>
</html>
```

3. You would also need to change the CSS

#pageheader becomes just *header*
#maincontainer becomes just *main*
#aside becomes just *aside*
#section becomes just *section*
#footer becomes just *footer*

## PART 6 – USING TEMPLATES

1. Now that we have index.html, we can use it as a template to construct the other 3 html files.

2. Copy index.html three more times and rename the files according to our navigation items, so enterweight.html, myweights.html, teamweights.html and winnerloser.html.

3. Now we need to restructure the code and content in each file, starting with the myweights.html file, this should allow someone to enter her name then see a list of all the weights and dates she has entered over time. For now this file will be just a dummy file. Remove everything between the **maincontent** div tags except the **h2** tag which we rename to "My Records". Also insert a pair of **form** tags.

```
<div id="maincontainer">
    <div id="maincontent">
            <h2>My Records</h2>
            <form>
            </form>
    </div>
```

4. Continue developing the **form** tags by inserting (a) a **label** and (b) an **input** box for the name to be found. Also (c) insert a **button** tag.

```
<div id="maincontainer">
    <div id="maincontent">
            <h2>My Records</h2>
            <form>
                    <label for="empName">Name</label>
                    <input id="empName" type="text" />
                    <button>Find my records</button>
            </form>
    </div>
```

5. Just after the closing **form** tag, add a few **p** or **div** tags to display dummy data

```
    <form>
            <label for="firstName">First Name</label>
            <input id="empName" type="text" ng-model="empName" />
            <button>Find my records</button>
    </form>
    <p>On [date] you weighed [empWeight] Kgs.</p>
    <p>On [date] you weighed [empWeight] Kgs.</p>
    <p>On [date] you weighed [empWeight] Kgs.</p>
    <p>On [date] you weighed [empWeight] Kgs.</p>
    <p>On [date] you weighed [empWeight] Kgs.</p>
</div>
<div id="aside">
```

6. Do something similar for enterweight.html, except we would need both the *name* and *weight*, so 2 fields. Also change the text between the **h2** tags.

```
<h2>Enter your weight</h2>
<form>
    <label for="empName">Your Name</label>
    <input id="empName" type="text" />
    <label for="empWeight">Your Weight Today</label>
    <input id="empWeight" type="text" />
    <button>Save Weight</button>
</form>
```

7. For teamweights.html simply replace the existing content we got from index.html with some dummy lines to represent records from our database

```
<div id="maincontainer">
    <div id="maincontent">
            <h2>Team Records</h2>
            <p>On [date] [empName] weighed [empWeight] Kgs.</p>
            <p>On [date] [empName] weighed [empWeight] Kgs.</p>
            <p>On [date] [empName] weighed [empWeight] Kgs.</p>
            <p>On [date] [empName] weighed [empWeight] Kgs.</p>
    </div>
    <div id="aside">
```

Notice that we actually print the date, name and weight. Also change the section title to "Team Records".

8. We wont be developing winner/loser right now, for now just delete **maincontent** area and replace with something like "future development".

## PART 7 – ADDING STYLES TO THE FORM

1. So far, two html files will use forms, the myweights.html file and enterweight.html.

2. Before we style these forms, it would be better to wrap each `label+input` pair of tags into a `container div`, so do that in the html first.

3. In myweights.html, add the `container div` to the label-input combination

```
<h2>My Records</h2>
<form>
    <div>
            <label for="empName">Name</label>
            <input id="empName" type="text" />
    </div>
    <button>Find my records</button>
```

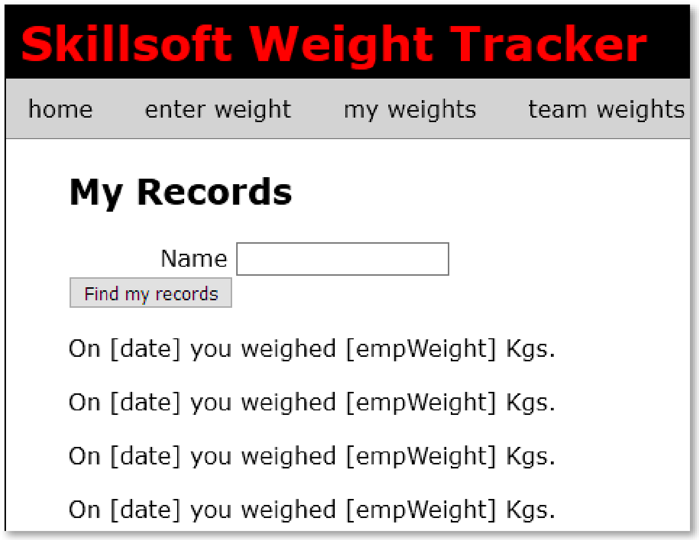
4. We would need to do this for the `button` as well.

```
<form>
    <div>
            <label for="empName">Name</label>
            <input id="empName" type="text" />
    </div>
    <div>
            <button>Find my records</button>
    </div>
</form>
```

5. The first style we apply is to have the label display as `inline-block`, give it an adequate `width` and align it's text to the right so all `labels` will end close to where the `input` boxes begin.

```
label{
    display:inline-block;
    width:110px;
    text-align:right;
}
```

6. That will put the `label` and `input` box next to each other but the `button` is still out of place, see image below.



My Records

On [date] you weighed [empWeight] Kgs.

On [date] you weighed [empWeight] Kgs.

On [date] you weighed [empWeight] Kgs.

On [date] you weighed [empWeight] Kgs.

7. For the `button`, you may need to play with the numbers but I found that if we apply a margin-left value we may be able to place the button just underneath where the input box starts

```
button{
    margin-left:115px;
    margin-top:10px;
}
```

8. We should apply the same technique to the form elements on the enterweight.html file, so wrap up the `label-input` combination inside of `div` tags as well as the `button`.

```
<form>
    <div>
            <label for="empName">Your Name</label>
            <input id="empName" type="text" />
    </div>
    <div>
            <label for="empWeight">Your Weight Today</label>
            <input id="empWeight" type="text" />
    </div>
    <div>
            <button>Save Weight</button>
    </div>
</form>
```

9. Look at the file in the browser and tweak the **width** property of the `label` tag in the style sheet so that the labels are all in one line. The following changes were made:

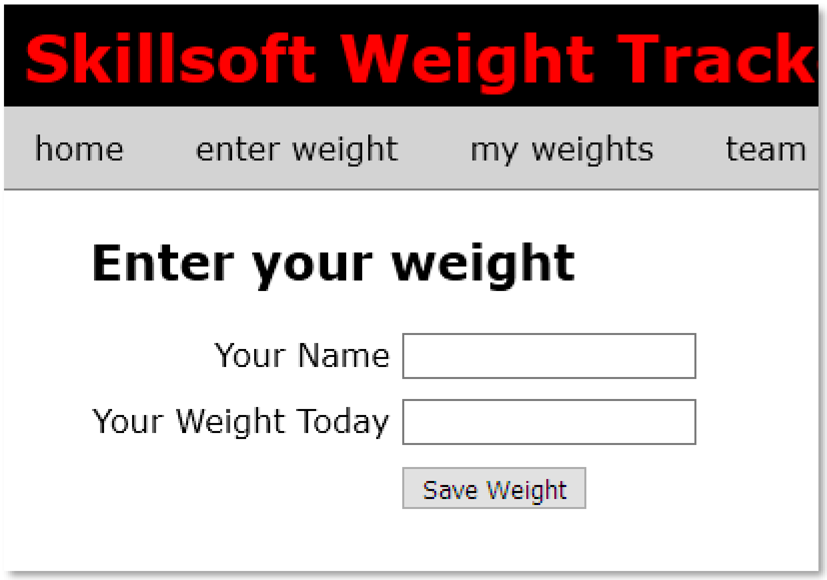
```
label{
    display:inline-block;
    width:150px;
    text-align:right;
}
button{
    margin-left:155px;
    margin-top:10px;
}
```

10. We should also separate the two sets of elements, so it would be better to do this with a `class` as we don't want to affect all `divs`. The following table box shows the class on the left and the html change to the enterweight.html on the right.

```
.formSeparator{
    margin-top:10px;
}
```

```
<form>
    <div>
            <label for="empName">Your
Name</label>
            <input id="empName" type="text"
/>
    </div>
    <div class="formSeparator">
            <label for="empWeight">Your
Weight Today</label>
            <input id="empWeight" type="text"
/>
    </div>
    <div>
```

11. The final look



Enter your weight

Your Name

Your Weight Today

12. The myweights.html file should be ok as well.

# Day02 JavaScript

## PART 1 – GETTING STARTED WITH JS

1. JS is good at manipulating the DOM, so let's test that theory out with a few scripts. On index.html, go to the **body** tag and insert the following JS code:

```
    </head>
    <body onload="alert('hello');">
        <div class="wrapper">
```

2. If that works, remove it from the browser and test it on one of the three **p tags** in the **maincontent** area, also change the event from *onload* to *onmouseover*:

```
    <div id="maincontent">
        <h2>How To Participate in the Program</h2>
        <p onmouseover="alert('hello');">Lorem ipsum dolor sit amet,
```

3. Create a new function in the head area using script tags, that will change the background colour of *any* element on the page:

```
    <script>
        function changeBgColor(el){
                el.style.backgroundColor='lightyellow';
        }
    </script>
```

   The **el** inside of the **changeBgColor()** function is so that the script knows which DOM element we need to change.

4. With the function created, we can now change the code in the **p** tag to call that function instead:

```
    <div id="maincontent">
        <h2>How To Participate in the Program</h2>
                <p onmouseover="changeBgColor(this);">Lorem ipsum
```

   Notice that when calling the function, we need to specify which element we need this function to be applied to, in this case it is the first **p** tag.

5. This works fine but the colour does not return to the original white background after the mouse moves away, we need to create a different function to handle that part:

```
    function removeBgColor(el){
        el.style.backgroundColor='';
    }
```

6. Now we can call this function to get rid of the yellow background but which event will we associate with this new function?

```
    <div id="maincontent">
        <h2>How To Participate in the Program</h2>
        <p onmouseover="changeBgColor(this);" onmouseout="removeBgColor(this);">
```

So as it turns out, there is a *onmouseout* event we can use to call **`removeBgColor()`**.

7. Doing this on all **`p tags`** we want this functionality is now as easy as copying and pasting

```
    <div id="maincontainer">
        <div id="maincontent">
                <h2>How To Participate in the Program</h2>
                <p onmouseover="changeBgColor(this);" onmouseout="removeBgColor(this);">Lorem…
                </p>
                <p onmouseover="changeBgColor(this);" onmouseout="removeBgColor(this);">Ut…
                </p>
                <p onmouseover="changeBgColor(this);" onmouseout="removeBgColor(this);">Ut…
                </p>
        </div>
```

8. The problem with this approach is that the html code is getting complex and there is not much separation of concerns going on. There is a better way if we implement a *listener* instead of accessing the DOM directly. Restructure the two functions to use listeners:
   First attempt to collect all the **`p tags`** in an array using **`getElementsByTagName`**:

```
    <script>
        let pTags = document.getElementsByTagName('p');
        function changeBgColor(el){
                el.style.backgroundColor='lightyellow';
        }
```

   This can be done before the start of the function inside of the script tags

9. Now create a loop to iterate through the array represented by the variable **`p tags`**:

```
    <script>
        let pTags = document.getElementsByTagName('p');
        for(let i = 0; i < pTags.length; i++) {

        }
        function changeBgColor(el){
```

10. In this case we have to add a listener and the type of event we wish to listen for, to each of the **`p tags`** in this collection, so add the following function to each tag.

```
    let pTags = document.getElementsByTagName('p');
    for(let i = 0; i < pTags.length; i++) {
        pTags[i].addEventListener('mouseover', function(){

        }
    }
```

   Notice that the **`addEventListener()`** method takes the *type of event* to listen to, *mouseover* in this case and also a function to run when that happens.

When the event fires on any **p** tag, in this case the mouseover event, we want to change the background colour, so we can add in that part using the code we used previously

```
    let pTags = document.getElementsByTagName('p');
    for(let i = 0; i < pTags.length; i++) {
        pTags[i].addEventListener('mouseover', function(){
                        this.style.backgroundColor='lightyellow';
        }, false);
    };
```

Notice that the **addEventListener()** method also takes a third parameter which I set as **false**. This parameter will affect whether the event (*mouseover* in this case) should fire in the capturing phase or in the bubbling phase. The default is bubble.

11. Remove the code that we had attached to the three **p tags** in index.html
Also remove the original two functions we had between the script tags.
Test the code, it may not work, this is because the entire DOM has not been loaded, so **move** the entire script with all its code to the bottom of the page, just above the ending **body** tag:

```
                        Copyright 2019. All rights reserved.
                </div>
        </div><!--closes the container div-->
        <script>
                let pTags = document.getElementsByTagName('p');
                for(let i = 0; i < pTags.length; i++) {
                        pTags[i].addEventListener('mouseover', function(){
                                this.style.backgroundColor='lightyellow';
                        }, false);
                };
        </script>
    </body>
```

Now the code will work one-way, we need to also cater for the **removal** of the style in the next step

12. Add a second block of code to add a different listener, this time the the *mouseout* event will be used and we will remove the background colour.

```
for(let i = 0; i < pTags.length; i++) {
    pTags[i].addEventListener('mouseover', function(){
                    this.style.backgroundColor='lightyellow';
    }, false);
    pTags[i].addEventListener('mouseout', function(){
                    this.style.backgroundColor='';
    }, false);
};
```

Note the code will also work for the `p` tag that is under "Health News".

In the next part we will start using an external .js file for our JavaScript.

## PART2 – TRANSFER TO EXTERNAL JS

1. Add a new directory to hold JS files, call it "scripts"

2. Create a new .js file inside of the new folder, call it scripts.js

3. Connect the new .js file with the html files, so in the head tag of index.html, add the following line:

```
<head>
            <title>Skillsoft Weight Tracker</title>
            <meta charset="utf-8"/>
            <link rel="stylesheet" type="text/css" href="styles/styles.css" />
            <script src="scripts/scripts.js"></script>
    </head>
    <body>
```

4. Remove all the code from between the `script` tags in the index.html file and paste it at the top of the scripts.js file. You can delete the **`script`** tags from the HTML document, we won't need it.

```
let pTags = document.getElementsByTagName('p');
for(let i = 0; i < pTags.length; i++) {
    pTags[i].addEventListener('mouseover', function(){
                    this.style.backgroundColor='lightyellow';
    }, false);
    pTags[i].addEventListener('mouseout', function(){
                    this.style.backgroundColor='';
    }, false);
};
```

**Refresh the html file, if it does not work, again remove the link to the .js file from the head area and put it just above the body tag.**

```
                    </div>
                </div><!--closes the container div-->
                <script src="scripts/scripts.js"></script>
        </body>
    </html>
```

## Part 3 – Adding a Dummy Landing Page

1. Prior to working with actual APIs, we need a dummy HTML page to land on when the form is submitted. We also need this to ensure that the JavaScript we add to the form is working properly.

   Copy any of the html files and remove the middle content area and just replace with some text indicating that the post was successful:

   ```
   <div id="maincontainer">
       <div id="maincontent">
               <h2>Thanks for your submission</h2>
               <p>Form submitted.
               </p>
       </div>
       <div id="aside">
   ```

   name this page, handler.html

2. Open enterweight.html and give a name to the form as well as an action value

   ```
   <h2>Enter your weight</h2>
   <form name="frmCollectWeights" action="handler.html">
       <div>
   ```

3. Add a connection to the scripts.js file from enterweight.html, copy the `script` tags from index.html and paste it into enterweight.html.

4. One of the advantages of having an external js file and developing code like this is being able to apply the same code in different situations.

   ```
   let pTags = document.querySelectorAll('p,input');
   for(let i = 0; i < pTags.length; i++) {
       pTags[i].addEventListener('mouseover', function(){
                   this.style.backgroundColor='lightyellow';
       }, false);
       pTags[i].addEventListener('mouseout', function(){
                   this.style.backgroundColor='';
       }, false);
   };
   ```

   Notice we used a slightly different function here, **`querySelectorAll()`** and we pass two parameters, so now both the **`input`** boxes **and** the **`p`** tags will have this behavior. This may or may not be what you want.

5. In our case, I want the background colour to change when the user has placed the mouse directly into the **`input`** box, so revert the code to the way it was before step 4 above.

6. Lets copy the entire structure we have for the **`p`** tag and paste it below but configure it for the **`input`** tag instead

```
          }, false);
     };
     //
     let inputTags = document.getElementsByTagName('input');
     for(let i = 0; i < inputTags.length; i++) {
          inputTags[i].addEventListener('mouseover', function(){
                              this.style.backgroundColor='lightyellow';
          }, false);
          inputTags[i].addEventListener('mouseout', function(){
                              this.style.backgroundColor='';
          }, false);
     };
```

7. Change the event listeners to `focus` and `blur` respectively

```
     let inputTags = document.getElementsByTagName('input');
     for(let i = 0; i < inputTags.length; i++) {
          inputTags[i].addEventListener('focus', function(){
                              this.style.backgroundColor='lightyellow';
          }, false);
          inputTags[i].addEventListener('blur', function(){
                              this.style.backgroundColor='';
          }, false);
     };
```

8. This code should work for the myweights.html file also, if we attach the scripts in the same way.

## Part 4 – Interrupting the form Submit Functionality

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:

```
        }, false);
    };
    //
    function validateForm(){

    }
```

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

```
    <div id="maincontent">
        <h2>Enter your weight</h2>
        <form name="frmCollectWeights" action="handler.html" onsubmit = "return(validateForm());">
                <div>
```

   Notice that the function name is the same in both files.

3. Insert a line into the `validateForm()` function to return false.

```
    function validateForm(){
        return false;
    }
```

   Try to submit the form, it should not go to the handler page.

4. 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.

## Part 5 – 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.

2. 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;
}
```

3. 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"];
    if (empName.value == "") {
```

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

## Part 6 – Adding Additional Validation

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 badName = /^([^0-9]*)$/;
          if (empName.value == "") {
                    document.getElementById("nameMessage").innerHTML="Name cannot be empty!";
           empName.focus();
           return false;
        }
     //
         if (!empName.value.match(badName)) {
           document.getElementById("nameMessage").innerHTML="Name cannot contain numbers!";
           empName.focus();
           return false;
        }
```

2. (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;
        }
```

3. 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:

```
        let pTags = document.getElementsByTagName('p');
        for(let i = 0; i < pTags.length; i++) {
                pTags[i].addEventListener('mouseover', function(){
                                this.style.backgroundColor='lightyellow';
                }, false);
                pTags[i].addEventListener('mouseout', function(){
                                this.style.backgroundColor='';
                }, false);
        };
        //
        let inputTags = document.getElementsByTagName('input');
        for(let i = 0; i < inputTags.length; i++) {
                inputTags[i].addEventListener('focus', function(){
                                this.style.backgroundColor='lightyellow';
                }, false);
                inputTags[i].addEventListener('blur', function(){
                                this.style.backgroundColor='';
                }, false);
        };
        //
        function validateForm(){
                document.getElementById("nameMessage").innerHTML="";
                document.getElementById("weightMessage").innerHTML="";
                let empName = document.forms["frmCollectWeights"]["empName"];
          let empWeight = document.forms["frmCollectWeights"]["empWeight"];
                let badName = /^([^0-9]*)$/;
                if (empName.value == "") {
                        document.getElementById("nameMessage").innerHTML="Name cannot be empty!";
            empName.focus();
            return false;
          }
        //
          if (!empName.value.match(badName)) {
            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;
        }
//
```

## PART 7 – CONNECTING TO THE APIS - THE OLD WAY

1. 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.

2. From the **`maincontent`** div, remove all the dummy text and just include a div to display the data and a button to call a function to get the data

```
<div id="maincontainer">
    <div id="maincontent">
            <div id="mainContent">
            <h2>Showing records for team</h2>
            <div id="records"></div>
            <button onclick="getData();">Get Records</button>
    </div>
    <div id="aside">
```

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

```
let pTags = document.getElementsByTagName('p');
let xmlhttp = new XMLHttpRequest();
let file = "json.txt";
function getData(){

}
```

4. In the function get ready to access the **`XMLHttpRequest`** object we just included in the file:

```
let file = "json.txt";
function getData(){
    xmlhttp.onreadystatechange = function() {

    }
}
```

Notice the **`onreaddystatechange`** property of the **`xmlhttp`** object takes a function, we will define that function now

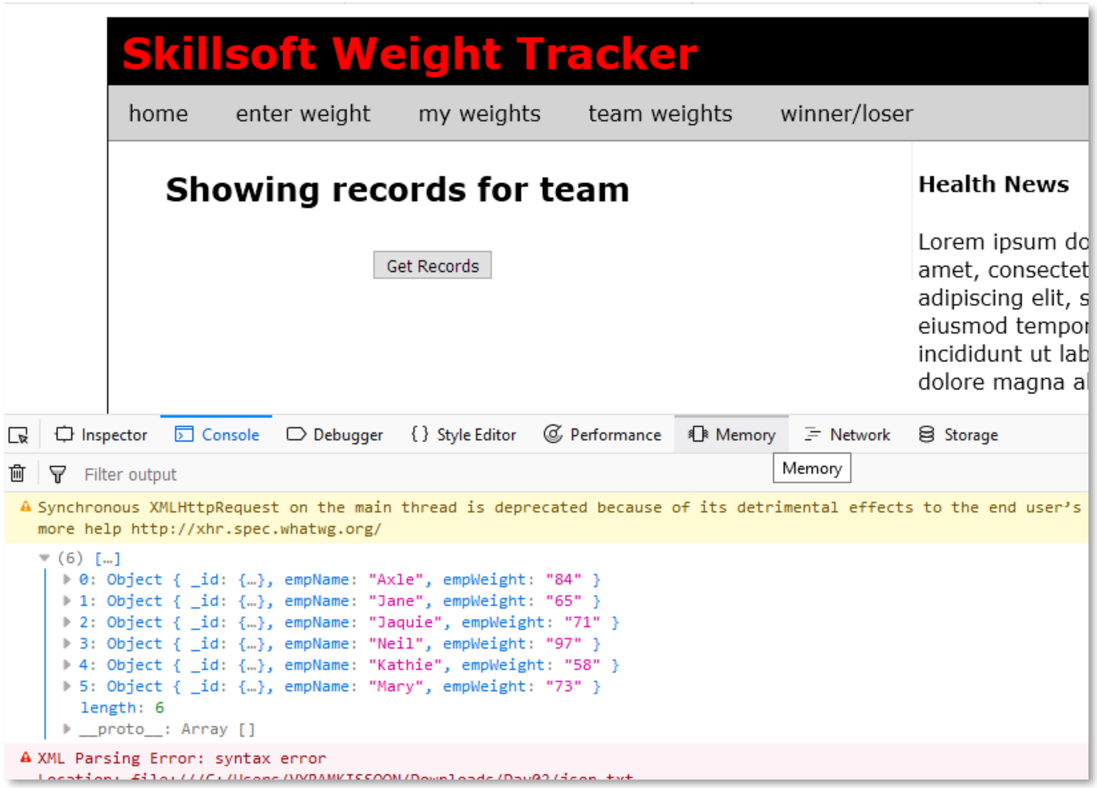
5. We first check the **`readyState`** of the **`onreadystatechange`** property and the status for specific numbers, if these are satisfied, we proceed to get the data via **`responseText`**

```
function getData(){
    xmlhttp.onreadystatechange = function() {
            if (this.readyState == 4 && this.status == 200) {
                                console.log(JSON.parse(this.responseText));
                        }
            }
}
```

6. After this we need to invoke the **`open()`** method as well as the **`send()`** method of the **`XMLHttpRequest`** object.

```
    function getData(){
        xmlhttp.onreadystatechange = function() {
                if (this.readyState == 4 && this.status == 200) {
                                console.log(JSON.parse(this.responseText));
                        }
                }
        xmlhttp.open("GET", file, false);
        xmlhttp.send(null);
    }
```

If all goes well, we should see the data in the **console** window of the browser You may see a parsing error, but this has to do with browser security, ignore it for now



```
Synchronous XMLHttpRequest on the main thread is deprecated because of its detrimental effects to the end user's
more help http://xhr.spec.whatwg.org/
(6) […]
  0: Object { _id: {…}, empName: "Axle", empWeight: "84" }
  1: Object { _id: {…}, empName: "Jane", empWeight: "65" }
  2: Object { _id: {…}, empName: "Jaquie", empWeight: "71" }
  3: Object { _id: {…}, empName: "Neil", empWeight: "97" }
  4: Object { _id: {…}, empName: "Kathie", empWeight: "58" }
  5: Object { _id: {…}, empName: "Mary", empWeight: "73" }
  length: 6
  __proto__: Array []
XML Parsing Error: syntax error
```

## Part 8 – The New Way

1. We now use the **`fetch()`** method to do the same thing we just did using the old **`ajax`** way. In the end, **`fetch()`** uses ajax in the background.

2. All we have to do is change the **`getData()`** function to this below. Everything else remains the same:

```
function getData(){
     fetch(file)
     .then(function(response){
             return response.json();
     })
     .then(function(jData){
             console.log(jData);
     });
}
```

It should show the same data in the **console** window. In this function, the **`fetch()`** method first returns a **promise** which is handled by the **`then()`** method that is chained to the **`fetch()`** method.
The first **`then()`** method extracts the data out of *response* and returns it as JSON to the next **`then()`** method. That second then method uses a function to pass JSON data back to the calling function, in this case it is the console window.

## Part 9 – 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.

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

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

```
function getData(){
    fetch(file)
    .then(function(response){
            return response.json();
    }).then(function(jData){
            displayData(jData);
    });
}
```

As usual, you may style the output as you wish.

## Part 10 – Using the APIs

1. In order to connect and consume the API, add the `url` address as a new variable in the scripts.js file. Then fetch the `url` instead of the file.

```
let xmlhttp = new XMLHttpRequest();
let file = "json.txt";
let url = "http://localhost:8000/getweights";
function getData(){
    fetch(url)
```

**Note, both the database and the node server file must be started for this to work.**
**The CORS plugin must be turned on to allow requests from the same domain.**

## PART 11 – USING THE ULTRA 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(url);
                const json = await response.json();
                displayData(json);
}
```

# Day03 jQuery and AngularJS

## Part 1 – Changing over from JS to jQuery

This day is broken up into two parts, **jQuery** and **AngularJS**. For both we will create new folders to store the part files in. *You could rename all files with a prepending system, so jq for jquery specific files and ng for AngularJS files*.

1. Create a new Folder called **Day03**, then inside of that folder create a new folder called **jQuery** and copy day02 files into that folder. We will do the same for AngularJS

2. To get started go to jQuery.com and find the most up to date CDN and insert it in the `head` tags between `script` tags. Do this on the index.html file.

```
        <head>
                <title>Skillsoft Weight Tracker</title>
                <meta charset="utf-8"/>
                <link rel="stylesheet" type="text/css" href="styles/styles.css" />
                <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
        </head>
```

3. Open the scripts.js file and remove all the coding from inside except the path to the json.txt file and the `url` pointing to `getweights`.

4. The first jQuery function we write is just to make sure the CDN is hooked up properly, so insert this function in the scripts.js file:

```
    //
    let file = "json.txt";
    let url = "http://localhost:8000/getweights";
     $(function(){
        alert("ready to rock!")
    });
```

5. If that works then we can test the *mouseover* effect we achieved with JS, in this case the code is much easier to write. Insert the following code between the function curly braces, comment the alert for now:

```
    let url = "http://localhost:8000/getweights";
     $(function(){
        //alert("ready to rock!")
        $("p").mouseover(function(){
        $(this).addClass("liYellow");
        });
    });
```

6. This code won't work as yet, we have to define the **`liYellow`** class in the CSS file, so open up styles.css and at the bottom add this style:

```
.liYellow{
    background-color:lightyellow;
}
```

Now you can test this code in the browser. Once again the background will change but not change back once the mouse is moved away from the **`p tag`**, so lets add to the function we started in #5 above:

```
$(function(){
    //alert("ready to rock!")
    $("p").mouseover(function(){
            $(this).addClass("liYellow");
    });
    $("p").mouseout(function(){
            $(this).removeClass("liYellow");
    });
});
```

## Part 2 – jQuery Validation

1. In this section, we will begin to use jQuery's power as a form validation technology. Using the enterweight.html file, import the **jQuery Validator Plugin** by attaching to a CDN and make sure jQuery is there also, but above the plugin:

```
        <meta charset="utf-8"/>
        <link rel="stylesheet" type="text/css" href="styles/styles.css" />
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
        <script src="https://cdn.jsdelivr.net/npm/jquery-
    validation@1.19.0/dist/jquery.validate.js"></script>
    </header>
    <nav>
```

2. Remove the **`span`** tags we inserted in Day02. They should be next to the input boxes. Also remove the **`onsubmit()`** function call within the **`form`** tags.

3. The **`button`** tag may need additional attributes and values, add an **`id`** and a **`type`** to the **`button`** tag:

```
    <div>
        <button id="sendData" type="submit">Save Weight</button>
    </div>
```

4. This particular validator also requires that the form elements have names, so add the **`name`** attribute to both fields and use the same **`id`** as the name

```
    <div>
        <label for="empName">Your Name</label>
        <input id="empName" name="empName" type="text" />
    </div>
    <div class="formSeparator">
        <label for="empWeight">Your Weight Today</label>
        <input id="empWeight" name="empWeight" type="text" />
    </div>
```

5. With the validator, we need to supply the form name, define rules for the form elements and messages. The last part of the **`validate()`** function is hooking up the **`form.submit()`** functionality.
   The first thing to do is to create a new function body and declare the form to be validated:

```
    $(function(){
        $("form[name='frmCollectWeights']").validate({});
    });
```

6. The **`validate()`** function takes a JSON structure, there are 3 main name-value pairs for the first layer, the first three *names* will take a json structure as their values:

```
    $(function(){
        $("form[name='frmCollectWeights']").validate({
                rules:{  },
                messages:{ },
                submitHandler:{ }
        });
    });
```

7. The **`submitHandler()`** function takes a function as its value, so we can complete that part of the code:

```
    $(function(){
        $("form[name='frmCollectWeights']").validate({
                rules:{},
                messages:{ },
                submitHandler:{
                        function(form){
                                form.submit();
                        }
                }
        });
    });
```

8. Now we can complete the first two name value pairs:

```
    $(function(){
        $("form[name='frmCollectWeights']").validate({
                rules:{empName:"required"},
                messages:{empWeight:"Weight is required" },
                submitHandler:{
                        function(form){
                                form.submit();
                        }
                }
        });
    });
```

Test the form now. Only the name field is working.

9. We need to add a rule and message for **empWeight**. We could just insert a comma after the first name-value pair and then insert the next field’s nam-value pair:

```
    $(function(){
        $("form[name='frmCollectWeights']").validate({
                rules:{empName:"required", empWeight:"required"},
                messages:{empName:"Name is required", empWeight:"Weight is required" },
                submitHandler:{
                        function(form){
                                form.submit();
                        }
                }
        });
    });
```

10. It would be better to structure the code like this, this way we can expand in the future without too much complications as the lines getting longer

```
        $("form[name='frmCollectWeights']").validate({
                rules:{
                        empName:"required",
                        empWeight:"required"
                },
                messages:{
                        empName:"Name is required",
                        empWeight:"Weight is required"
                },
                submitHandler:{
                        function(form){
                                form.submit();
                        }
                }
```

11. With this new structure in place, we can expand to even other types of checks such as ensuring only letters in the name field and only numbers in the weight field.
This will create a new layer in the json structure.
In this next level, **empName** and **empWeight** become the start of the next level and required in the second level with the value of true or false:

```
    rules:{
        empName:
        {
                required:true,
                minlength:3
        },
        empWeight:{
                required:true,
                digits:true
        }
    },
    messages:{
```

12. The validator does not have a standard validation method for ensuring an alphabetic character field only, so we can add additional methods. At the bottom of the scripts.js file add this method to check for alphabetic characters only:

```
        $(function(){
                jQuery.validator.addMethod("lettersOnly", function (value, element) {
                        return this.optional(element) || /^[a-zA-Z]+$/i.test(value);
                }, "Please enter letters only.");
        });
```

13. With the new method in place, we can add a new rule, we do not need a message because the message is already in the new method:

```
    $("form[name='frmCollectWeights']").validate({
        rules:{
                empName:
                {
                        required:true,
                        minlength:3,
                        lettersOnly:true
                },
                empWeight:{
                        required:true,
```

## PART 3 – JQUERY API FUNCTIONALITY

In this section we will use jQuery to interact with out APIs. Add the jQuery CDN and the jQuery Validation Plugin to all the other HTML pages in the jQuery Folder.

1. teamweights.html already has a `button`. Just change the button’s *onclick* attribute to `id` instead. jQuery will add a listener for when this button is clicked. Also remove the () from the end of `getData`.

```
        <h2>Showing records for team</h2>
        <div id="records"></div>
        <button id="getData">Get Records</button>
    </div>
    <div id="aside">
```

2. Start setting up the function with the shell:

```
    //
    let file = "json.txt";
    let url = "http://localhost:8000/getweights";
    $(function() {
       $("#getData").click(function(){

         });
    });
```

3. Next add the `ajax()` method in the middle

```
    $(function() {
       $("#getData").click(function(){
                   $.ajax({});
         });
    });
```

4. Then fill in the name-value pairs, with just the name part first:

```
    $(function() {
       $("#getData").click(function(){
                   $.ajax({
                             url:
                             dataType:
                             success:
                             error:
                   });
         });
    });
```

5. Fill in the values but notice that *success* and *error* both take functions as their values:

```
    $(function() {
       $("#getData").click(function(){
          $.ajax({
                             url:"json.txt",
                             type:"GET",
                             dataType:"json",
                             success:function(result){
                                       console.log(result);
                             },
                             error:function(err){
                                       console.log(err);
                             }
                   });
       });
    });
```

6. If this work, we can either add the old JS file from day02 or copy the `displayData()` function from that file to this new JS file. Since the file names are the same, we will copy the function from

```
                   return this.optional(element) || /^[a-zA-Z]+$/i.test(value);
         }, "Please enter letters only.");
    });
    //
    function displayData(dataArray){
        let htmlOut = "";
        for(let i=0; i<dataArray.length; i++){
                htmlOut+=dataArray[i].empName + " weighed " + dataArray[i].empWeight + " Kgs<br />";
        }
        document.getElementById("records").innerHTML=htmlOut;
    }
```

7. Finally back in the **getData()** function remove **console.log()** and add **displayData()** instead:

```
            $.ajax({
                        url:"json.txt",
                        type:"GET",
                        dataType:"json",
                        success:function(result){
                                displayData(result);
                        },
```

## Part 4 – Changing over from JS to AngularJS

This day is broken up into two parts, jQuery and AngularJS. For both we will create new folders to store the part files in. *You could rename all files with a prepending system, so jq for jquery specific files and ng for AngularJS files*.

1. Create a new Folder called Day03, then inside of that folder create a new folder called **AngularJS** and copy day02 files into that folder.

2. To get started go to https://angularjs.org/ and find the most up to date CDN and insert it in the **head** tags between **script** tags. Insert the link to scripts.js also. Do this on the teamrecords.html file.

```
        <meta charset="utf-8"/>
        <link rel="stylesheet" type="text/css" href="styles/styles.css" />
        <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.5/angular.min.js"></script>
    </head>
```

*There should already be a link to our JS file at the bottom of this html document.*

3. Open the scripts.js file and remove all the coding from inside except the path to the json.txt file and the **url** pointing to **getweights**. Also leave the **displayData()** function. The entire file at this point should look like this:

```
    //
    let file = "json.txt";
    let url = "http://localhost:8000/getweights";
    //
    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;
    }
    //
```

4. The first Angular task is to declare an **`app`**, which we will later use to add a **`controller`** to, so insert this line in the scripts.js file:

```
    let file = "json.txt";
    let url = "http://localhost:8000/getweights";
    //
    let app = angular.module('SkillsApp', [] );
```

5. Once we have an **`app`** object, we can use it to invoke the **`controller`** method, pass it a name for our controller as well as the **`$scope`** object. Use the **`$scope`** object within the function body to declare a variable and display it in the console window.

```
    let url = "http://localhost:8000/getweights";
    //
    let app = angular.module('SkillsApp', [] );
    app.controller('Weights', function($scope) {
        $scope.data = "Skillsoft";
        console.log($scope.data);
    });
    //
    function displayData(arr) {
```

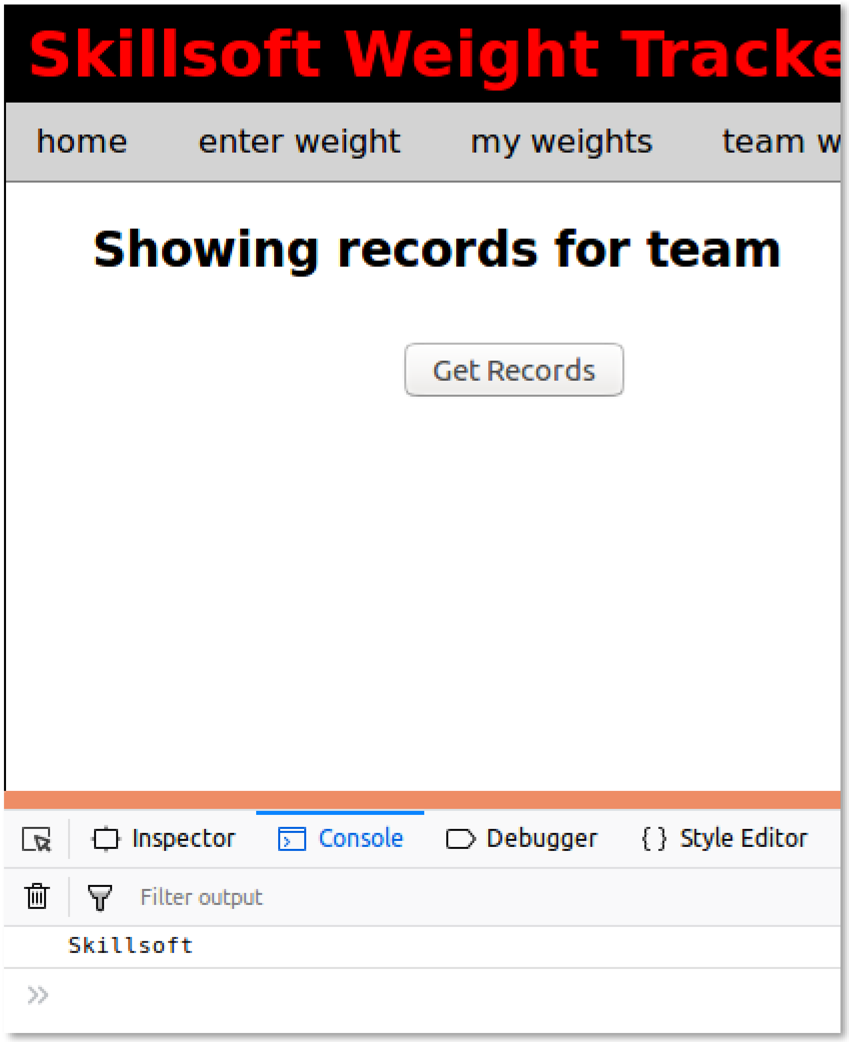
6. Back to teamweights.html, add the Angular **`app`** to the **`body`** tag as a directive.

```
        <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.5/angular.min.js"></script>
    </head>
    <body ng-app="SkillsApp">
        <div class="wrapper">
```

7. Add the **`controller`** to the **`maincontent`** div so it becomes available to that section of the code.

```
    </nav>
    <div id="maincontainer">
        <div id="maincontent" ng-controller="Weights">
                <h2>Showing records for team</h2>
```

8. Refresh the teamweights.html file, you should see Skillsoft in the console window. This indicates that Angular is hooked up and working properly.



Showing records for team

```
Skillsoft
```

## PART 5 – ANGULAR’S HTTP SERVICE

With the **app** and **controller** in place, we can now use Angular’s HTTP service to read the text file as well as the api.

1. Continue to build the controller, add the **$http** service as the second parameter to the function’s parameter list:

```
let app = angular.module('SkillsApp', [] );
app.controller('Weights', function($scope, $http) {
    $scope.data = "Skillsoft";
    console.log($scope.data);
});
```

2. Wrap the next lines into the **get()** method of the **http service**. Pass the **file** variable to the **get()** method.

```
app.controller('Weights', function($scope, $http) {
    $http.get(file){
            $scope.data = "Skillsoft";
            console.log($scope.data);
    };
});
```

3. The **get()** method actually returns a promise, that promise must be handled with a **then()** method chained onto the **get()** method. At the same time we can pass into the **then()** method, the **response** object from the call to the **url** or **file**.

```
app.controller('Weights', function($scope, $http) {
    $http.get(file).then(function(response){
            $scope.data = "Skillsoft";
            console.log($scope.data);
    });
});
```

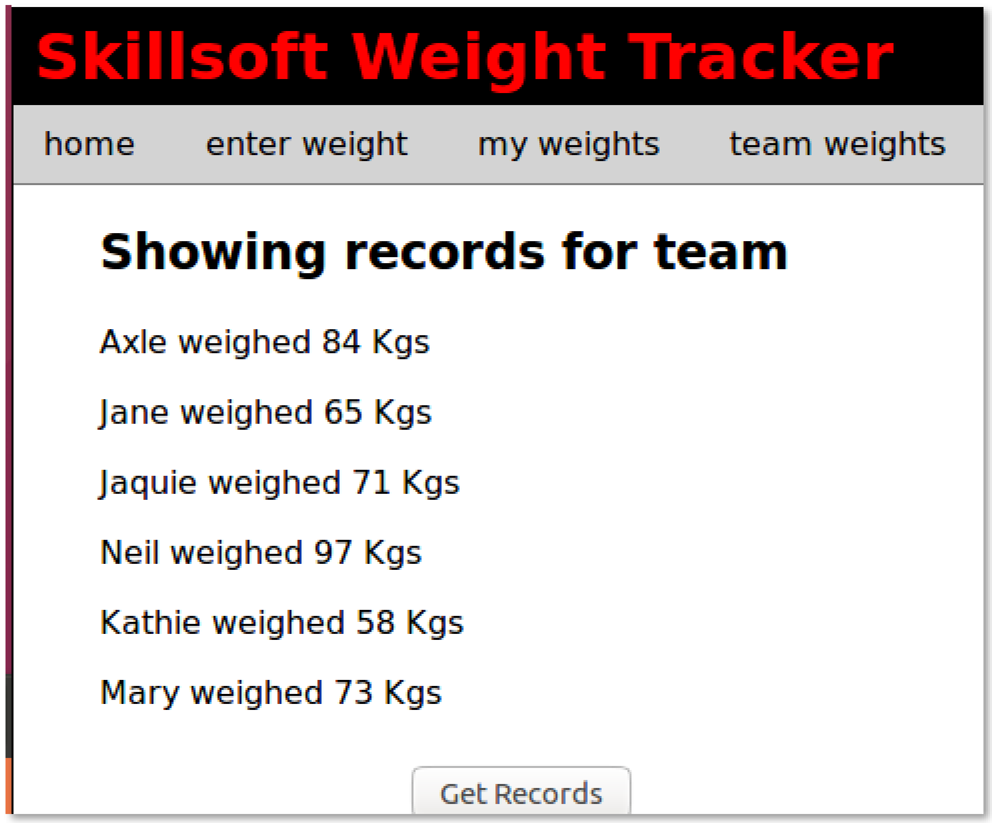
4. Pass the response object returned from the call to the **file**, to **$scope.data** and test again

```
app.controller('Weights', function($scope, $http) {
    $http.get(file).then(function(response){
            $scope.data = response;
            console.log($scope.data);
    });
});
```

5. We really only need the **data** part of the **response** object, so extend response to **response.data** and pass that to the **displayData()** method we wrote in day02.

```
let app = angular.module('SkillsApp', [] );
app.controller('Weights', function($scope, $http) {
    $http.get(file).then(function(response){
            displayData(response.data);
    });
});
```

Test the application now



Showing records for team

Axle weighed 84 Kgs

Jane weighed 65 Kgs

Jaquie weighed 71 Kgs

Neil weighed 97 Kgs

Kathie weighed 58 Kgs

Mary weighed 73 Kgs

## Part 6 – Using Angular's ng-repeat directive

Up to now we did not change the way the data is displayed, however Angular does have some powerful features we can use, like the **`ng-repeat`** directive.

1. Go back to teamrecords.html and remove the button entirely, then move the closing **`div`** tag down 3 spaces.

```
<div id="maincontainer">
<div id="maincontent" ng-controller="Weights">
    <h2>Showing records for team</h2>
    <div id="records">



    </div>
</div>
```

2. Insert a line between the **`div`** tags:

```
<h2>Showing records for team</h2>
<div id="records">
    {{emp.empName}} weighed in at {{emp.empWeight}} Kgs
</div>
</div>
<div id="aside">
```

3. Next complete the **`div`** tag with the Angular directive **`ng-repeat`**. We will define **`allWeights`** in the JS file

```
<h2>Showing records for team</h2>
<div id="records" ng-repeat="emp in allWeights">
    {{emp.empName}} weighed in at {{emp.empWeight}} Kgs
</div>
</div>
<div id="aside">
```

4. Back in scripts.js, instead of passing the data to the **`displayData()`** function, pass it to a **`$scope`** variable called **`$scope.allWeights`**.

```
let app = angular.module('SkillsApp', [] );
app.controller('Weights', function($scope, $http) {
    $http.get(file).then(function(response){
            $scope.allWeights = response.data;
    });
});
```

5. Remove the entire **`displayData()`** function. Refresh the html file, it should now show the data like before, but now more efficiently.

## PART 7 – POSTING NEW DOCUMENTS WITH ANGULARJS

So far we have just been reading data that has already been uploaded to the api but in this segment we will use Angular's http service to post new data, create new documents etc.

1. Open the enterdata.html file in your editor and wire it up just like we did for teamweights.html.

```
        <title>Skillsoft Weight Tracker</title>
        <meta charset="utf-8"/>
        <link rel="stylesheet" type="text/css" href="styles/styles.css" />
        <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.5/angular.min.js"></script>
    </head>
    </head>
    <body ng-app="SkillsApp">
        <div class="wrapper">
                <img id="logo" src="images/chart.gif" />
                <header>
                        <h1>
                                <a href="index.html">Skillsoft Weight Tracker</a>
                        </h1>
                </header>
                <nav>
                        <ul>
                                <li><a href="index.html">home</a></li>
                                <li><a href="enterweight.html">enter weight</a></li>
                                 ...
                        </ul>
                </nav>
                <div id="maincontainer">
                        <div id="maincontent"  ng-controller="Weights">
                                <h2>Enter your weight</h2>
                                <form name="frmCollectWeights" action="handler.html" onsubmit =
    "return(validateForm());">
```

*There should already be a link to the JS file at the bottom of this html document.*

2. Remove the **onsubmit()** function from the **form** tag and use Angular's directive **ng-submit**. We will point that directive to a function we will define shortly called **frmSubmit()**. Also remove the action attribute and it's value.

```
    <div id="maincontent"  ng-controller="Weights">
        <h2>Enter your weight</h2>
        <form name="frmCollectWeights" ng-submit="frmSubmit()">
                <div>
```

3. We also need to wire up the **input** boxes so that our **controller** knows when values change, remove any **span** tags if there are any:

```
    <div>
        <label for="empName">Your Name</label>
        <input id="empName" type="text" ng-model="empName"/>
    </div>
    <div class="formSeparator">
        <label for="empWeight">Your Weight Today</label>
        <input id="empWeight" type="text" ng-model="empWeight"/>
    </div>
    <div>
```

the html file is now complete

4. With the html in place we can turn our attention to the JS, open scripts.js and enter the following code within the curly braces of the `controller()` method:

```
    app.controller('Weights', function($scope, $http) {
        $http.get(file).then(function(response){
                $scope.allWeights = response.data;
        });
    //
    $scope.frmSubmit = function(){};

    });
```

5. This will handle the form submit directly to the api, but to complete the function body, we will need to supply it with the `$http` service and later add at least four name-value pairs:

```
    //
        $scope.frmSubmit = function(){
                $http({});
        };
    });
```

6. Complete the http method with the four names:

```
    //
        $scope.frmSubmit = function(){
                $http({
                        method  : ,
                        url     : ,
                        headers : ,
                        data    :
                });
        };
    });
```

7. Now fill in the values:

```
        $scope.frmSubmit = function(){
                $http({
                        method  : 'POST',
                        url     : 'http://localhost:8000/putweights',
                        headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
                        data    : 'empName='+$scope.empName + '&empWeight='+$scope.empWeight
                });
```

Note that for this part, the json.txt file **will not work**, it does not have a handler for posting, we will need to use the API for this.

8. Before testing make sure the following:
   a. the `mongodb` service is running

    b. the **api** is running at localhost
    3. The **CORS** client is turned on

9. In a regular browser or the REST client, go to the endpoint **getweights** to see a list of all the **documents** currently in the **collection**.
10. Enter a new **document** and then refresh the browser to see if the record has entered the database

# Day04 Installing Node, MySQL & MongoDB

## Part 1 – Installing NodeJS, MySQL and MongoDB

This day is all about installation and configuration, **MySQL**, **MongoDB** and **NodeJS**. In all cases, we will be working with **Ubuntu 16** and the latest software of these three technologies.

## Installing nodejs on Ubuntu 16

**These are the commands we will be entering (in order) to get node.js installed**

Install curl

```
sudo apt install curl
```

Use curl to make sure we are downloading to a stable node PPA

```
curl -sL https://deb.nodesource.com/setup_10.x | sudo bash -
```

With the PPA downloaded, we can then run the command to install nodejs itself:

```
sudo apt-get install -y nodejs
```

Check the version of node

**node –v (**should return v10.10.0**)**

Check the version of npm, which is installed automatically

**npm –v (**should return 6.4.1 or newer**)**

Check the installation by running the following command:

```
sudo node http_server
```

This should return something like *Debugger listening on ws://127.0.0.1:9229/a...*

# Install MySQL on Ubuntu

Run the following commands to install MySQL on Ubuntu

```
sudo apt-get update

sudo apt-get install mysql-server
```

After installation is complete, the mysql_secure_installation utility runs, which prompts for the mysql root password and other security stuff

Start the service

```
sudo systemctl start mysql
```

Confirm that the service is running

```
sudo service mysql status
```

To launch the shell

```
/usr/bin/mysql -u root –p
```

Note: do not enter the password after the –p, leave it blank,

The system will ask for password on the next line, see arrow

Confirm in shell by verifying that the **mysql >** prompt is present

Create a database

```
CREATE DATABASE Weights;
```

To see the db just created

```
SHOW DATABASES;
```

Create a new user if necessary

```
CREATE USER 'axle'@'localhost' IDENTIFIED BY ‘1234’;

GRANT ALL PRIVILEGES ON *.* TO 'axle'@'localhost' IDENTIFIED BY '1234';
```

[This is an alternative**: GRANT ALL PRIVILEGES ON *.* TO 'axle'@'localhost' IDENTIFIED BY 'axle';]**

```
FLUSH PRIVILEGES;
```

Check if user is in the database

```
SELECT user FROM mysql.user;
```

Give admin rights

```
GRANT ALL PRIVILEGES ON Weights.* to axle@localhost;
FLUSH PRIVILEGES;
```

Exit the mysql shell and log in as the new user created (\q)

```
/usr/bin/mysql -u axle –p
```

## Working with databases and tables (10)

Change the database to weights2 and create a new table using the following code:

```
use weights;

CREATE TABLE EmployeeWeights (
id INT(4) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
empName VARCHAR(30) NOT NULL,
empWeight FLOAT(5, 1) NOT NULL,
email VARCHAR(50),
reg_date TIMESTAMP
);
```

Verify that the table exist

```
show tables;
```

verify that the columns match what was entered

```
SHOW COLUMNS FROM EmployeeWeights ;
```

After looking at it we realize that weight should be float 4, 1 instead.
Alter one of the columns

```
ALTER TABLE EmployeeWeights  MODIFY empWeight FLOAT(4,1);
```

(changing from 5,1)

Verify change:

```
SHOW COLUMNS FROM EmployeeWeights ;
```

Test the database and table

```
INSERT INTO EmployeeWeights  (empName, empWeight) VALUES ('Axle', 55.8);
```

Then Select all from table

This ends the pure MySQL part, we still have to configure MySQL to work with nodejs

## INSTALLATION OF MONGODB ON UBUNTU

1. Import the MongoDB public GPG key:

   **sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 9DA31620334BD75D9DCB49F368818C72E52529D4**

2. Create a list file for MongoDB in the path /etc/apt/sources.list.d/

   **echo "deb [ arch=amd64 ] https://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/4.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-4.0.list**

3. Reload local package database

   **sudo apt-get update**

4. Install a stable MongoDB package

   **sudo apt-get install -y mongodb-org**

5. Check the version to make sure installation is complete:

   **mongod --version**

Parts 6 and 7 may not be necessary, the installation usually creates its own directory and may be different depending on how Mongo was installed.

6. Create a directory to store databases (from root)
   **sudo mkdir –p /data/db/**

7. Make sure the current user has ownership of this directory
   **sudo chown `id –u` /data/db**

   NB: The file **/etc/mongodb.conf will have a line that points to the default directory**

8. Start the service
   **sudo mongod**
   Service should start in the terminal window; you would need a second widow to interact with MongoDB
9. In a second terminal run the command

```
sudo mongo  (enter linux password)
```

10. At the command prompt > type in *db* to see the current test database

    **CTRL-C** to exit the terminal

## Part 2 – Configuring a New User in MySQL

1. Create a database

```
CREATE DATABASE Weights;
```

2. To see the db just created

```
SHOW DATABASES;
```

3. Create a new user and grant all permissions

```
CREATE USER 'axle'@'localhost' IDENTIFIED BY ‘1234’;
GRANT ALL PRIVILEGES ON *.* TO 'axle'@'localhost' IDENTIFIED BY '1234';
FLUSH PRIVILEGES;
```

4. Check that the user exists:

```
SELECT user FROM mysql.user;
```

5. Give admin rights to the new user

```
GRANT ALL PRIVILEGES ON Weights.* to axle@localhost;
FLUSH PRIVILEGES;
```

6. Exit the MySQL shell and log in as the new user (\q to quit)

```
/usr/bin/mysql -u axle –p
```

1. Change the database to weights2 and create a new table using the following code:

```
use weights;

CREATE TABLE EmployeeWeights (
id INT(4) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
empName VARCHAR(30) NOT NULL,
weight FLOAT(5, 1) NOT NULL,
email VARCHAR(50),
reg_date TIMESTAMP
);
```

2. Verify that the table exist

```
show tables;
```

3. Verify that the columns match what was entered

```
SHOW COLUMNS FROM EmployeeWeights ;
```

4. Enter a record

```
INSERT INTO EmployeeWeights  (empName, empWeight) VALUES ('Axle', 55.8);
```

5. Verify the record.

```
SELECT * FROM EmployeeWeights;
```

[this part is optional]

6. After looking at it we realize that weight should be float 4, 1 instead. Alter one of the columns

```
ALTER TABLE EmployeeWeights  MODIFY empWeight FLOAT(4,1);
```

7. Verify the change

```
SHOW COLUMNS FROM EmployeeWeights ;
```

## PART 4 – MYSQL AND NODEJS

1. Now Create a folder in which to work and CD into that folder.
   Run `npm install` within that folder to create a package.json file. Install the mysql driver for node

   ```
           npm install express mysql
   ```

2. We create a server file, using `touch`, inside the folder eg http_server.js.
   Enter the connection code below:

   ```
           let mysql = require('mysql');
           let connection = mysql.createConnection({
             host: 'localhost',
             user: 'axle',
             password: '1234',
             database: 'Weights'
       });
   ```

3. It In the next block of code, attempt to connect to the database (**the mysql service must be running**)

   ```
           connection.connect(function(err) {
            if (err) {
             return console.error('error: ' + err.message);
            }
             console.log('Connection successful');
           });
   ```

   Run the file using **`node http_server`** and check the console log

4. The following function will retrieve records. Replace the code (second block) you entered above or change the necessary lines:

   ```
           connection.connect(function(err) {
                   if (err) throw err;
                   if(connection.query("SELECT * FROM EmployeeWeights", function (err, result) {
                           if (err) throw err;
                           console.log(result);
                           })
                   ){connection.end();};
           });
   ```

**Or this code may be easier to work with:**

```
connection.connect(function(err) {
        if (err) throw err;
        if(connection.query("SELECT * FROM EmployeeWeights", function (err, result) {
                if (err) throw err;
                console.log(result);
                })
        );
        connection.end();
});
```

5. Now try inserting a new record. Insert this block of code above the block from #4 above so that when the record is inserted, it will be read at the same time.

```
connection.connect(function(err) {
        if (err) {
                 return console.error('error: ' + err.message);
        }
        let sql = "INSERT INTO EmployeeWeights(empName, empWeight) VALUES(  'Sally', 59.9
)";
        if(connection.query(sql, function (err, result) {
                if (err) throw err;
                console.log(result);
                })
        );
        connection.end();
});
```

Here is the entire file to insert a new record called Sally and then read all reacords:

```
let mysql = require('mysql');
let connection = mysql.createConnection({
   host: 'localhost',
   user: 'axle',
   password: '1234',
   database: 'Weights'
});
//
connection.connect(function(err) {
    if (err) {
             return console.error('error: ' + err.message);
    }
    let sql = "INSERT INTO EmployeeWeights(empName, empWeight) VALUES(
'Tommy', 79.4 )";
    if(connection.query(sql, function (err, result) {
            if (err) throw err;
            console.log(result);
            })
    );
    if(connection.query("SELECT * FROM EmployeeWeights", function (err, result)
{
            if (err) throw err;
            console.log(result);
            })
    );
    connection.end();
});
```

## PART 5 – MYSQL AND NODEJS API

[This section is optional]

Time permitting, we can demonstrate the way MySQL works with node in the development of an API. For this use **nodefiles_mysq**l folder which contains all the neessary changes to work with MySQL. Same files are zipped in the file nodefiles_mysql.zip.

If you decide to show node's connection to mysql delivering the same details as with MongoDB then the following changes must be made in the three main files, routes.js, models.js and controllers.js.

routes.js

For routes, no changes except that we will create just one route:

```
'use strict';
module.exports = function(app) {
var weights = require('../controllers/weightController');
app.route('/getweights').get(weights.getweights);
};
```

models.js

In the models file, we just need to establish a connection and export it.

```
let mysql = require('mysql');
let connection = mysql.createConnection({
        host:"localhost",
        user:"axle",
        password:"1234",
        database:"Weights"
});
//
connection.connect(function(err) {
  if (err) throw err;
});
module.exports = connection;
```

controllers.js

Controllers is where most of the logic goes. **`Getweights()`** is an exported function. Inside the function, declare and execute the SQL, deal with any errors and of course, return to the client, any records found:

```
const Weight = require('../models/emp_weights');
//get records
exports.getweights = function(req,res){
        sql="SELECT * FROM EmployeeWeights";
        if(Weight.query(sql, function(err, result){
                if(err){
                        return console.error('error:' +err.message);
                };
                res.json(result);
        }));
};
//
```

There should not be any changes in the http_server file.

Before testing this code, make sure that **`npm install`** is run or use npm to install mysql.

## PART 6 – CREATING MONGODB DATABASES AND COLLECTIONS

Before proceeding, delete any `Weights` database that is currently in the system. Do a show dbs and all databases will be shown. To delete a db, just use it then issue the command **`db.dropDatabase()`**.

Also make sure you are in the MongoDB directory under Day04

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

```
use Weights;
```

2. Add a collection

```
db.createCollection(“EmployeeWeights”)
```

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

```
db.EmployeeWeights.find()
```

4. Enter a record

```
db.EmployeeWeights.insertOne( {empName : “Joe”,  empWeight : 55.6 })
```

5. Verify the record.

```
db.EmployeeWeights.find()
```

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

```
db.EmployeeWeights.insertOne( {empName : “mary”,  empWeight : 65.9 })
```

7. Verify the new record

```
db.EmployeeWeights.find()
```

8. Lets change (update) Joe’s record:

```
db.EmployeeWeights.update(
   {empName : "Joe"},
   {$set: {empWeight : 56.5 } }
)
```

9. Verify the change

```
db.EmployeeWeights.find()
```

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

```
db.EmployeeWeights.insertOne(
{
empName : "Sally",
empWeight : 65.9,
Date : new Date()
}
)
```

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

```
db.EmployeeWeights.find().pretty()
```

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

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

## PART 7 – INTEGRATING WITH NODEJS

1. Make sure that you are in the MongoDB directory inside of **Day04**.
   Run **`npm init`** within that folder to create a package.json file. Accept the defaults except for name, use http_server.js.

```
            npm init
```

2. Run **`npm install mongodb`** to install the mongodb package.

```
            npm install mongodb
```

3. We create a server file, using `touch`, inside the folder eg http_server.js.
   Enter the connection code below:

```
            const mClient = require('mongodb').MongoClient;
            const url = "mongodb://localhost:27017/";
            mClient.connect(url, {useNewUrlParser:true}, function(err, db){
                        if(err) throw err;
                        console.log("Connected to mongodb");
                        db.close();
            } );
```

4. It In the next block of code, attempt to connect to the collection

```
        const mClient = require('mongodb').MongoClient;
        const url = "mongodb://localhost:27017/";
        mClient.connect(url, {useNewUrlParser:true}, function(err, db){
                if(err) throw err;
                let wdb=db.db("Weights");
                wdb.collection("EmployeeWeights").findOne(
                   {"empName":"Joe"},
                   function(err, result){
                        if(err) throw err;
                        console.log(result);
                   });
                db.close();
        } );
```

Note: in the code above, we can change this line to show the weight for joe eg
`console.log(result.empWeight);`

5. Try to find all the documents in the collection. You may think that just executing the find method without any name value pairs may work, but it doesent

```
        const mClient = require('mongodb').MongoClient;
        const url = "mongodb://localhost:27017/";
        mClient.connect(url, {useNewUrlParser:true}, function(err, db){
                if(err) throw err;
                let wdb=db.db("Weights");
                wdb.collection("EmployeeWeights").find(
                   {},
                   function(err, result){
                        if(err) throw err;
                        console.log(result);
                   });
                db.close();
        } );
```

Well it returns then entire server structure, but in the middle of all that data is our weights informtion, we just need to extract it, see below.

6. Just chain the **`toArray()`** method onto the **`find()`** method

```
const mClient = require('mongodb').MongoClient;
const url = "mongodb://localhost:27017/";
mClient.connect(url, {useNewUrlParser:true}, function(err, db){
        if(err) throw err;
        let wdb=db.db("Weights");
        wdb.collection("EmployeeWeights").find({}).toArray(
           function(err, result){
                if(err) throw err;
                console.log(result);
           });
        db.close();
} );
```

7. Here is the entire code:

```
const mClient = require('mongodb').MongoClient;
const url = "mongodb://localhost:27017/";
mClient.connect(url, {useNewUrlParser:true}, function(err, db){
        if(err) throw err;
        let wdb=db.db("Weights");
        wdb.collection("EmployeeWeights").find({}).toArray(
                function(err, result){
                        if(err) throw err;
                        console.log(result);
                });
        db.close();
} );
```

8. Lets now try to insert a new document using our server js file. We will use the **`insertOne()`** method, but first create a variable to represent the json object that will represent our new document.

```
const mClient = require('mongodb').MongoClient;
const url = "mongodb://localhost:27017/";
mClient.connect(url, {useNewUrlParser:true}, function(err, db){
        if(err) throw err;
        let wdb=db.db("Weights");
        let newEmployee = {empName:"Johan", empWeight: 86.7};
        wdb.collection("EmployeeWeights").find({}).toArray(function(err, result){
                if(err) throw err;
                console.log(result);
        });
        db.close();
} );
```

9. We will leave the **`find()`** method in place as we would want to verify that Johan was inserted, but we can copy and past the **`find()`** method and replace find with **`insertOne()`**

```
                if(err) throw err;
                let wdb=db.db("Weights");
                let newEmployee = {empName:"Johan", empWeight: 86.7};
                wdb.collection("EmployeeWeights").insertOne(
                    newEmployee, function(err, result){
                        if(err) throw err;
                        console.log("Inserted one document");
                });
                wdb.collection("EmployeeWeights").find({}).toArray(function(err, result){
```

10. Insert a record with a **`date`** field

```
        if(err) throw err;
                let wdb=db.db("Weights");
                let newEmployee = {
                                empName:" Harry",
                                empWeight: 51.5,
                                date: new Date(Date.now()).toISOString()
                        };
                wdb.collection("EmployeeWeights").insertOne(newEmployee, function(err,
        result){
```

When you execute the http_server file, you see Johanes is inserted but the date value is more readable.

Here is the entire http_server.js file so far:

```
const mClient = require('mongodb').MongoClient;
const url = "mongodb://localhost:27017/";
mClient.connect(url, {useNewUrlParser:true}, function(err, db){
        if(err) throw err;
        let wdb=db.db("Weights");
        let newEmployee = {
                empName:'Harry',
                empWeight: 96.4,
                date: new Date(Date.now()).toISOString()
        };
        wdb.collection("EmployeeWeights").insertOne(
                newEmployee, function(err, results){
                        if(err) throw err;
                        console.log("Inserted one document");
                });
        wdb.collection("EmployeeWeights").find({}).toArray(
                function(err, result){
                        if(err) throw err;
                        console.log(result);
                });
        db.close();
} );
```

11. Perform a search with a criteria.
we can modify the current code to create a search term

```
        let wdb=db.db("Weights");
        let searchTerm = {
                        empName:"Harry"
                };
        wdb.collection("EmployeeWeights").insertOne(newEmployee, function(err,
result){
```

12. Also change the method from **`insertOne()`** to **`findOne()`**, then pass into **`findOne()`** the **`searchTerm`** object to be found.

```
        wdb.collection("EmployeeWeights").findOne(
                searchTerm,
                function(err, result){
                        if(err) throw err;
                        console.log(result);
        });
```

Here is the entire code, with the last block commented out

```
        const mClient = require('mongodb').MongoClient;
        const url = "mongodb://localhost:27017/";
        mClient.connect(url, {useNewUrlParser:true}, function(err, db){
                if(err) throw err;
                let wdb=db.db("Weights");
                let searchTerm = {
                                empName:"Harry"
                };
        //
                wdb.collection("EmployeeWeights").findOne(
                        searchTerm,
                        function(err, result){
                                if(err) throw err;
                                console.log(result);
                });
                db.close();
        } );
```

**Optional**

**`find()`** will return an array so we have to chain the **`toArray()`** method onto find and deal with any error or result:

```
                wdb.collection("EmployeeWeights")
                .find(searchTerm)
                .toArray(
                        function(err, result){
                                if(err) throw err;
                                console.log(result);
                        }
                );
```

Entire file using **`find()`**

```
        const mClient = require('mongodb').MongoClient;
        const url = "mongodb://localhost:27017/";
        mClient.connect(url, {useNewUrlParser:true}, function(err, db){
                if(err) throw err;
                let wdb=db.db("Weights");
                let searchTerm = {
                                empName:"Harry"
```

```
		};
	//
		wdb.collection("EmployeeWeights")
		.find(searchTerm)
		.toArray(
			function(err, result){
				if(err) throw err;
				console.log(result);
			}
		);
		db.close();
	} );
```

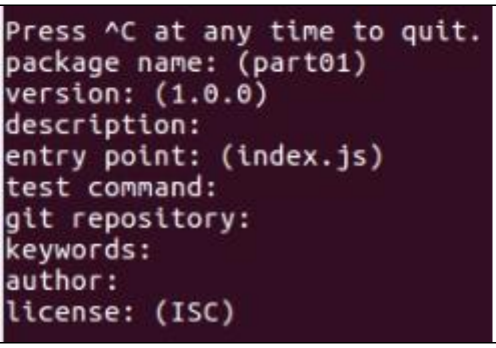
# Day05 Building the APIs

## PART 1 – VERIFY NODEJS

1. Create a folder called **`Day05`** and inside of that folder, create a new folder called **`Part01`**.

2. Open a terminal inside of **`Part01`** and run the command **`npm init`**

3. Follow the prompts and just hit **`enter`** for each question, this is just to create a package.json file



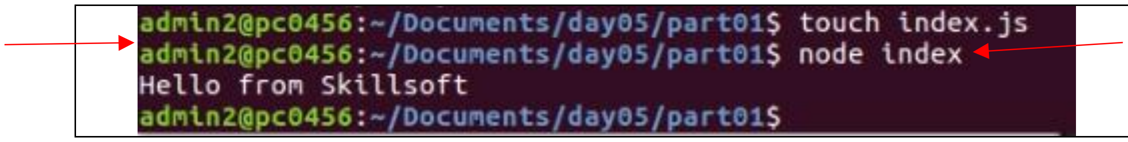
```
Press ^C at any time to quit.
package name: (part01)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
```

4. According to the .json file, node will look for index.js in order to execute the code inside, so use touch to create index.js inside of the **`Part01`** folder.

5. Add the following code to execute. This is just to make sure that node is working and it is executing properly.

```
console.log("Hello from Skillsoft!");
```

6. Execute index.js by typing in the command **`node index`** from the command prompt. It should show "Hello from Skillsoft". This step confirms that we can move on to other parts.



```
admin2@pc0456:~/Documents/day05/part01$ touch index.js
admin2@pc0456:~/Documents/day05/part01$ node index
Hello from Skillsoft
admin2@pc0456:~/Documents/day05/part01$
```

## PART 2 – BUILDING A SIMPLE NODEJS APP

1. Create a folder called **`Part02`** inside of Day05.

2. Open a terminal inside of **`Part02`** and run the command **`npm init`**

3. Follow the prompts and just hit **`enter`** for each question, this is just to create a package.json file. This time instead of accepting index.js, change it to http_server.js.

4. Open a terminal window pointing to the **`Part02`** folder and using touch, create a .js file called http_server.js.

5. At this point, you should have two directories inside of **`Day05`**, **`Part01`** and **`Part02`**. Inside of **`Part02`** you should have two files, http_server.js and package.json.

6. Open http_server.js inside of a text editor and type the following lines:

```
    const http = require('http');
    const hostname = "localhost";
    const port = 8000;
```

This code means that we are using the http module of **`nodejs`**, and we will define the other two parameters that the http service requires.

7. Next we will define a variable to point to the **`createServer()`** method which will hold a reference to the server

```
    const SkillServer = http.createServer();
```

**A special note on the http.createServer() method**.

The createServer() method returns a web server object, which will listen for requests and then handle those requests by returning responses to the client, which could be a browser.

createServer() takes a function that is called each time a request is made.
Once a request is made and that request gets to the server, it is considered a request object and it is based on an HTTP method or verb. The headers object also exist on that request, but it is a separate object.

There are some requests that need special handling, such as POST and PUT. These need special handlers that can work with the ReadableStream interface. When the incoming data happens to be string, then it is possible to handle this string data as an array.

The response object on the other hand is an instance of the ServerResponse class. It is a WritableStream. To send back a response to the client means dealing with the stream methods such as write() and end().

8. The createServer() method takes a function that handles both the request and response objects. Extend the method to include that function as an anonymous function.

```
const myServer = http.createServer(function(request, response){

});
```

9. This now gives us access to these two objects, so we can interrogate the `request` object for things like parameter values or form values and we can use the `response` object to send data back to the client. In this case we will only use the response object to send an ok as well as some text to the client

```
const myServer = http.createServer(function(request, response){
    response.writeHead(200, {'Content-Type':'text/plain'});
    response.write("Hello from Skillsoft");
    response.end();
});
```

10. Finally we can call the listen method and pass it the port and hostname

```
mySerer.listen(port, hostname);
```

Here is the entire http_server.js file

```
const http = require('http');
const hostname = "localhost";
const port = 8000;

const myServer = http.createServer(function(request, response){
        response.writeHead(200, {'Content-Type':'text/plain'});
        response.write("Hello from Skillsoft");
        response.end();
});

mySerer.listen(port, hostname);
```

11. In a browser navigate to http://localhost:8000 and you should see the message from the `response.write()` method call.

## PART 3 – INCLUDING AND WORKING WITH NODE PACKAGES

1. Copy the folder called **`Part02`** paste it inside of Day05 and then rename it to **`part03`**.

2. Open a terminal inside of **`Part03`** and run the command **`npm install`** which will install everything that part02 had, it will use the json file from that directory.

3. While still in part03, install body parser by running this command from a terminal window that is pointing to Part03 directory: **`npm install body-parser --save`**

4. Install express by running this command from a terminal window that is pointing to Part03 directory: **`npm install express --save`**

5. Open the http_server.js and replace the first line with this one

```
const http = require('express');
const hostname = "localhost";
const port = 8000;
```

6. Next we will require body parser, and express does not need the hostname, so you could remove it or leave but do not use it

```
const http = require('express');
const http = require('body-parser');
const port = 8000;
```

7. Create a new variable and point it to the constructor of express

```
const http = require('express');
const http = require('body-parser');
const port = 8000;
const app = express();
```

8. This example will use the POST method of the browser to pass data to our server, so we need body-parser to help with the identification of form values:

```
const http = require('express');
const http = require('body-parser');
const port = 8000;
const app = express();
app.use(bodyParser.urlencoded({extended:false}));
```

9. At this point we can use the the **app** object again to call the **post()** method. That **post()** method takes a route to send the request to and a function that handles the request and response objects.

```
const http = require('express');
const http = require(`body-parser');
const port = 8000;
const app = express();
app.use(bodyParser.urlencoded({extended:false}));
app.post('/addnewdoc', function(request, response){});
```

With this code in place, we can use it to now get values from a form. For example on the form there is a field called **empName.** We can get the value that the user put into that field by interrogating the **body** property of the **request** object.

```
app.post('/addnewdoc', function(request, response){
   let empName = request.body.empName;


});
```

10. We can extend this to the **weight** value as well. Also for now lets just use the log to show that we did receive those values on the server end

```
app.post('/addnewdoc', function(request, response){
   let empName = request.body.empName;
   let empWeight = request.body.empWeight;
   console.log(`POST success, you sent ${empName} and ${empWeight}, thanks!`);
   response.end(`POST success, you sent ${empName} and ${empWeight}, thanks!`);
});
```

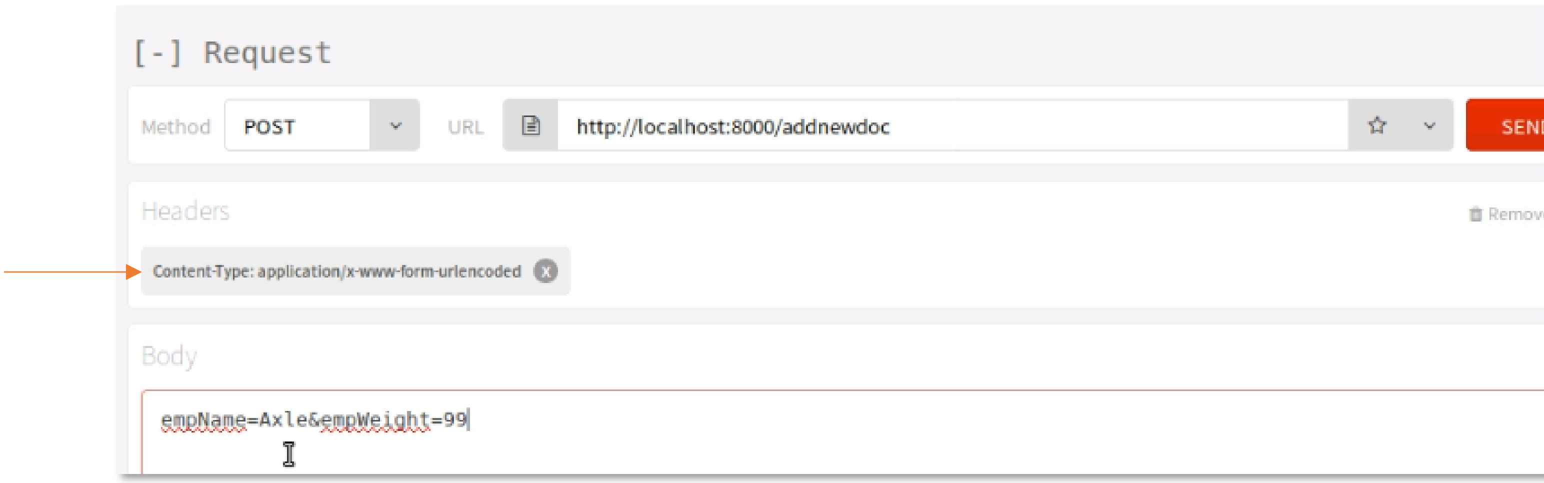
11. Finally for this file, remember when we ran the server file using node and the terminal window, we did not get any response. Lets change the **listen** method to use **app** and also to inform the developer that the service has started. This code goes to the bottom of the server file.

```
app.listen(port, function(){
     console.log("Listening " + port);
});
```

Here is the entire file, so far:

```
const express = require('express');
const bodyParser = require('body-parser');
//const hostname = "localhost";
const port = 8000;
const app = express();
app.use(bodyParser.urlencoded({extended:false}));
//
app.post('/addnewdoc', function(request, response){
   let empName = request.body.empName;
   let empWeight = request.body.empWeight;
   console.log(`POST success, you sent ${empName} and ${empWeight}, thanks!`);
   response.end(`POST success, you sent ${empName} and ${empWeight}, thanks!`);
});
//
app.listen(port, function(){
     console.log("Listening " + port);
});
```

12. Now we have to test this out using a REST client in the browser, see below Remember to turn on CORS and pass along a header


[-] Request
Method
POST
URL
http://localhost:8000/addnewdoc
SEN
Headers
Remov
Content-Type: application/x-www-form-urlencoded
Body

```
empName=Axle&empWeight=99
```

13. Check the response in the terminal window and also on the browser's console window.

## PART 4 – ROUTING BASICS

1. Copy the folder called **`Part03`** paste it inside of **`Day05`** and then rename it to **`part04`**.

2. Run **`npm install`** using a terminal window.

3. Erase most of the code except for the first 5 lines and the listener at the bottom, so your http_server.js file in part06 should look like the code below:

```
const express = require('express');
const bodyParser = require('body-parser');
const port = 8000;
const app = express();
app.use(bodyParser.urlencoded({extended:false}));
//
app.listen(port, function(){
    console.log("Listening " + port);
});
```

4. Next we will install *router*, so run this code in the terminal window that points to part04: **`npm install router –save`**

5. Create a new variable and point it to the **`Router()`** constructor from **`express`**:

```
const app = express();
app.use(bodyParser.urlencoded({extended:false}));
//
const router = express.Router();
```

6. We now have router to construct routes and the first route is going to be the **root route**. Each route will use a method that represents an http REST verb. The first parameter will be the route path and the second will be a function that handles the request and response objects.

```
const app = express();
app.use(bodyParser.urlencoded({extended:false}));
//
const router = express.Router();
router.get('/', function(req, res){
    res.send("You are on the root route");
});
```

7. Before we can run this code, we need to tell our express app, to use `router` for executing routes. The `app.use()` method is saying to use router once you get to the root of this server path

```
router.get('/', function(req, res){
    res.send("You are on the root route");
});
//
app.use('/', router);
//
app.listen(port, function(){
```

8. Spin the application and go to a browser and the root address which remember is being served from port 8000, you should see "You are on the root route". We can now proceed to build other routes.

9. Create an "About Us" route by copying the `get()` route and replacing the first parameter with something like "/aboutus".

```
router.get('/', function(req, res){
    res.send("You are on the root route");
});
//
router.get('/aboutus', function(req, res){
    res.send("You are on the about us  route");
});
//
app.use('/', router);
```

**NOTE: whenever we make a change on the server code, we must stop and start the application, unless we use nodemon.**

10. Continue to include other routes as necessary, here is the entire file. Note that the last route gets a specific document, based on some parameter passed in via the **URL**.

```
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();
router.get('/', function(req, res){
    res.send("You are on the root route");
});
//
router.get('/aboutus', function(req, res){
    res.send("You are on the about us  route");
});
//
router.get('/employees/:employeeID', function(req, res){
    res.send("You are viewing employee # " + req.params.employeeID);
});
//
app.use('/', router);
//
app.listen(port, function(){
    console.log("Listening " + port);
});
```

## PART 5 – DECOMPOSING ROUTES

1. Copy the folder called **Part04** paste it inside of Day05 and then rename it to **part05**.

2. Open a terminal inside of **Part05** and run the command **npm install** which will install everything that **part04** had, it will use the **JSON** file from that directory.

3. Create a new folder called **routes** and inside of that directory, create a new .js file called routes.js.

4. The first line will be a variable pointing to a function, we have to do this in order for other files in our application to know that the routes file exists.

```
module.exports = function(app){};
```

Also notice that we have to pass the Express app into this function as a parameter, so that it becomes available to the entire function.

5. Next we will CUT the three get() functions from our http_server.js file into this one

```
module.exports = function(app){
    router.get('/', function(req, res){
            res.send("You are on the root route");
    });
//
    router.get('/aboutus', function(req, res){
            res.send("You are on the about us  route");
    });
//
    router.get('/employees/:employeeName'', function(req, res){
            res.send("You are viewing employee# " + req.params.employeeName'');
    });
//
};
```

6. However this file does not have access to **router**, it has access to **app**, which has access to **router**, so change the router object to the app object.

```
module.exports = function(app){
    app.get('/', function(req, res){
            res.send("You are on the root route");
    });
//
    app.get('/aboutus', function(req, res){
            res.send("You are on the about us  route");
    });
//
    app.get('/employees/:employeeName'', function(req, res){
            res.send("You are viewing employee " + req.params.employeeName'');
    });
//
};
```

7. Back in the http_server file, we have to let it know where to find routes.js, so create a variable and point it to the new routes.js file inside of the routes directory.

```
Const app = express();
const router = express.Router();
const routes = require('./routes/routes');
```

Remember we had cut the three route functions, so this file should be very short.

8. Use the newly created **routes** object to register the Express **app** via it's constructor

```
const router = express.Router();
const routes = require('./routes/routes');
routes(app);
//
app.use('/', router);
```

The rest of the http_server.js file remain unchanged.

Here is the entire http_server.js file, the routes.js file follows:

```
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.use('/', router);
//
app.listen(port, function(){
    console.log("Listening " + port);
});
```

routes.js

```
module.exports = function(app){
    app.get('/', function(req, res){
            res.send("You are on the root route");
    });
//
    app.get('/aboutus', function(req, res){
            res.send("You are on the about us  route");
    });
//
    app.get('/employees/:employeeName'', function(req, res){
            res.send("You are viewing employee " + req.params.employeeName'');
    });
//
};
```

9. Test the application, it should work just like before, no changes. But we have now ported our routes into a separate file, making future changes easier

## PART 6 – DECOMPOSING CONTROLLERS

1. Create a new directory called **`controllers`** and create a new **`.js`** file called controller.js

2. Open the controller.js file in an editor and start entering the first controller function. Remember controllers will take responsibility for making several decisions. The first controller should handle what happens when the user navigates to the root route:

```
exports.getdefault = function(req, res){
  res.send('You are on the root route.');
};
```

In this case we are not exporting the entire file, but each function is exported individually

3. Continue to develop this file by completing all the route functions, in other words, write functions that match the routes we had before. For now these functions are very simple, but in **day06**, they will become a bit more complicated.

```
exports.getdefault = function(req, res){
   res.send('You are on the root route.');
};
//
exports.aboutus=function(req, res){
   res.send('You are on the about us route.');
};
//
exports.employees=function(req, res){
   res.send('You are viewing employee# ' + req.params.employeeName);
};
//
exports.getallrecords=function(req, res){
   res.send('You are on the getallrecords route.');
};
```

   I have just added a new function **`getallrecords`** to do some interacting with the Weights database soon. This is the entire controller.js file so far.

4. Back in the routes.js file, we need to let this file know that there is a controller handling each route, so basically routes.js is now acting like a pointer to a controller function, which does the final piece in deciding what to serve to the client. Delete all the code and replace it with just this one line for now.

```
module.exports = function(app){
     let controller = require('../controllers/controller');
};
```

5. We can now expand on this file to include matching functions for each route

```
module.exports = function(app){
   let controller = require('../controllers/controller');
   app.route('/').get(controller.getdefault);
   app.route('/aboutus').get(controller.aboutus);
   app.route('/employees/: employeeName').get(controller.employees);
   app.route('/getallrecords').get(controller.getallrecords);
   //
};
```

   Notice that each line represents a route. That route now points to a function in the controller file.

6. Test the application, it should work just like in **`part06`**. Note, there is nothing to do in the http_server.js file.

## PART 7 – INTEGRATING AND WORKING WITH MONGODB

1. Copy the folder called **`Part05`** paste it inside of Day05 and then rename it to **`part06`**.

2. Open a terminal inside of **`Part06`** and run the command **`npm install`** which will install everything that **`part05`** had, it will use the JSON file from that directory.

3. While still in **`part06`**, install MongoDB by running this command from a terminal window that is pointing to Part06 directory: **`npm install mongodb --save`**

4. Open the http_server.js and replace the first line with this one, **remove everything else**.

```
const MongoClient = require('mongodb').MongoClient;
```

5. Next we will create another variable and make it equal to the address of where mongo lives

```
const MongoClient = require('mongodb').MongoClient;
const url = 'mongodb://localhost:27017';
```

6. Create a new variable and have it represent the database we created in mongo on day4

```
const MongoClient = require('mongodb').MongoClient;
const url = 'mongodb://localhost:27017';
const dbName = 'Weights';
```

7. Create a new variable and have it point to the **`MongoClient`** constructor. The constructor takes 2 parameters, a url of where mongo lives and a json object that is directly out of the documentation.

```
const MongoClient = require('mongodb').MongoClient;
const url = 'mongodb://localhost:27017';
const dbName = 'Weights';
const mClient = new MongoClient(url, {
    useNewUrlParser: true
});
```

9. At this point we can verify that we can connect to the database using the connect() method of mClient. Pass it a function which takes the error object.

```
mClient.connect(function(err) {
    if(err) console.log("Error");
  console.log("Success!!!");
  mClient.close();
});
```

10. If you see a "success!!!" Message in the terminal window, then we are able to connect to **mongodb**

## PART 8 – CRUD OPERATIONS WITH MONGODB

1. Copy the folder called **Part06** paste it inside of Day05 and then rename it to **part07**.

2. Open a terminal inside of **Part07** and run the command **npm install** which will install everything that **Part06** had, it will use the json file from that directory.

3. This file will insert a new document into the **Weights** database. Open the http_server.js file, which is basically the same file from **Part06**. Then add, below the existing code, the following **variable** which points to **function**. The function takes two **parameters**, the first represents our **Weights** database and the second is a **function** that will run, after the **putDocuments()** function is ran.

```
    const putDocuments = function(db, callback) {

    };
```

4. Next we will add a variable to represent the collection inside of our database. So the database is represented by **db** and the **EmployeeWeights** collection is represented by **collection**.

```
    const putDocuments = function(db, callback) {
       const collection = db.collection('EmployeeWeights');
    };
```

5. Use the **insertMany()** function of the **collection** object to insert 2 documents

```
    const putDocuments = function(db, callback) {
       const collection = db.collection('EmployeeWeights');
       collection.insertMany([
          {"empName":"Axle", "empWeight" : "85.8"},
          {"empName":"John", "empWeight" :"102"}
        ]);
    };
```

6. Although this code will work, we should also supply a second parameter to the **insertMany()** method which is a function and it will handle any errors as well as feedback from the server, after the insert.

```
const putDocuments = function(db, callback) {
   const collection = db.collection('EmployeeWeights');
   collection.insertMany([
      {"empName":"Axle", "empWeight" : "85.8"},
      {"empName":"John", "empWeight" :"102"}
    ], function(err, result) {
      console.log("Inserted 2 documents/records");
      callback(result);
    });
};
```

7. Also the second parameter includes a callback. That will send execution back to the caller function which we will change next.

8. With the **putDocuments()** function complete, go back to the **connect()** function and call **putDocuments**(). This function should be in the http_server.js file already from part 04. But first we need to create an variable that represents our database object. **mClient** in our case has a **db** method to which we pass the database name. We are returned with an object representing the **Weights** database.

```
mClient.connect(function(err) {
     if(err) console.log("Error");
     const db = mClient.db(dbName);
    putDocuments(db, callback);
});
```

9. We no need to replace the word **callback** with an actual anonymous function, which will execute the **mClient.close()** function, thereby closing the database connection. So make sure the **mClient.close()** function is wrapped inside of the anonymous function.

```
const db = mClient.db(dbName);
putDocuments(db, function(){
  mClient.close();
});
```

10. Before executing the code, show the current database with the current set of documents. Then run this new code to show that the 2 documents were inserted.

Here is the entire code so far:

```
const MongoClient = require('mongodb').MongoClient;
const url = 'mongodb://localhost:27017';
const dbName = 'Weights';
const mClient = new MongoClient(url, {
     useNewUrlParser: true
});
//
mClient.connect(function(err) {
     if(err) console.log("Error");
     const db = mClient.db(dbName);
   putDocuments(db, function(){
              mClient.close();
     });
});
//
const putDocuments = function(db, callback) {
   const collection = db.collection('EmployeeWeights');
   collection.insertMany([
      {"empName":"Axle", "empWeight" : "85.8"},
      {"empName":"John", "empWeight" :"102"}
    ], function(err, result) {
      console.log("Inserted 2 documents/records");
      callback(result);
    });
};
```

## PART 9 – SETTING UP MONGOOSE

1. Copy the folder called `Part07` paste it inside of Day05 and then rename it to `part08`.

2. Open a terminal inside of `Part08` and run the command `npm install` which will install everything that `part07` had, it will use the json file from that directory.

3. We also need to install **Mongoose**, so once inside of `Part08` run this command: `npm install mongoose –save.` Mongoose is an ORM which interacts with the `Weights` database and abstracts away much of the annoyances of working directly with the database natively.

4. Create a new directory called `models` and touch a new .js file inside of models called models.js and add the following lines.

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/Weights', { 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.

5. Next we will define the schema.

```
const wSchema = new mongoose.Schema({
  empName: String,
  empWeight: String
});
```

6. Although the schema will work as is, and a collection is created by default, we will be using a collection called **`EmployeeWeights`**, so lets make sure the schema knows this:

```
const wSchema = new mongoose.Schema({
  empName: String,
  empWeight: String
},{
      collection:'EmployeeWeights'
});
```

7. Finally for the models.js file, we need to export our schema

```
module.exports = mongoose.model('Weights', wSchema);
```

8. Here is the entire file

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/Weights', { useNewUrlParser: true });
const wSchema = new mongoose.Schema({
  empName: String,
  empWeight: String
},{
      collection:'EmployeeWeights'
});
module.exports = mongoose.model('Weights', wSchema);
```

At this point, test the application to make sure there are no errors, so test all the endpoints

## PART 10 – EXPANDING THE CONTROLLER FUNCTIONS TO WORK WITH DATABASE

1. Copy the folder called `Part08` paste it inside of Day05 and then rename it to `part09`.

2. Open a terminal inside of `Part09` and run the command `npm install` which will install everything that `part08` had, it will use the json file from that directory.

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

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

4. Next we will expand the `getallrecords` function. That function will use the `Weight` variable created above and its attached `find()` method. Delete the `res.send()` function or comment it out.

```
exports.getallrecords=function(request, response){
    Weight.find({}, function(err, results){});
    //res.send('You are on the getallrecords route.');
};
```

5. The `find()` method will handle any errors and any returns from the query, so lets expand on it.

```
    exports.getallrecords=function(request, response){
        Weight.find({}, function(err, results){
                if (err)
                    response.end(err);
                response.json(results);
        });
        //res.send('You are on the getallrecords 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.

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

```
app.route('/getallrecords').get(controller.getallrecords);
```

7. Test the code by opening a browser and navigating to `http://localhost:8000/getallrecords`

**Optional**

8. We can now try to get a single record by passing in the *name* to get in the url. Remember we had a route called **`employees`** and a controller called **`employees`**. Expand the **`employees()`** controller method to find an employee by her name:

```
        exports.employees = function(req, res) {
            let empToFind = req.params.employeeName;
            Weight.find({empName:empToFind}, function(err, results){
               if (err)
                   res.end(err);
                          res.json(results);
               });
        };
```

9.  Add a route to the routes.js file

```
  app.route('/employees/: employeeName').get(controller.employees);
```

10.        Test the code by opening a browser and navigating to **`http://localhost:8000/employees/Axle`**

## PART 11 – EXPANDING THE CONTROLLER TO DELETE FROM DATABASE

1. Copy the folder called **`Part09`** paste it inside of **`Day05`** and then rename it to **`part10`**.

2. Open a terminal inside of **`Part10`** and run the command **`npm install`** which will install everything that **`part09`** had, it will use the JSON file from that directory.

3. In the routes.js file, copy any of the previous route lines and change the route to be **`deletebyname`**.

```
  app.route('/deletebyname/:byname').delete(controller.deletebyname);
```

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

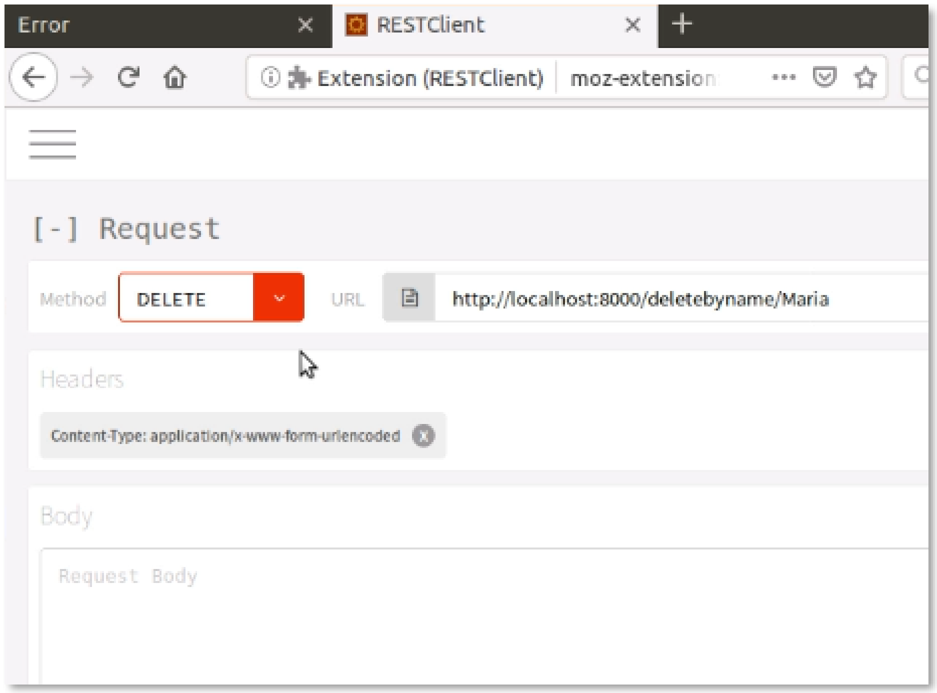
4. Create a matching function in the controller.js file, in fact we can just copy, paste and edit the **getbyname()** function. Jut change **find()** to **deleteOne()** and create a new variable to hold the name to be deleted.

```
exports.deletebyname = function(req, res) {
  let delName = req.params.byname;
  Weight.deleteOne({empName:delName}, function(err, result) {
    if (err)
      res.send(err);
    res.end(`Deleted ${delName}`);
  });
};
```

In this function, we get the name to delete from the **URL**, store it in a variable, then pass the variable as a value to the **deleteOne()** method. If no errors we send a text message to the client.

5. We can now try to delete a single record by passing in the *name* to get in the **URL**. We will need to use the **REST** client. Remember to change the method to **DELETE**. Also **CORS** must be turned on.

**Note the function is looking for empName, so if the document was not stored with that name/value type of structure, the delete will fail.**



Method: DELETE
URL: http://localhost:8000/deletebyname/Maria
Content-Type: application/x-www-form-urlencoded

**Note the function is looking for `empName`, so if the document was not stored with that name/value type of structure, the delete will fail.**

## PART 12 – EXPANDING THE CONTROLLER TO ADD A NEW DOCUMENT TO THE DATABASE

1. Copy the folder called **`Part10`** paste it inside of Day05 and then rename it to **`part11`**.

2. Open a terminal inside of **`Part11`** and run the command **`npm install`** which will install everything that **`part10`** had, it will use the json file from that directory.

3. In the routes.js file, copy any of the previous route lines and change the route to be **`putdoc`**.

```
app.route('/putdoc').post(controller.putdoc);
```

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

4. Create a matching function in the controller.js file, in fact we can just copy, paste and edit the **`deletebyname()`** function.

```
    exports.putnewdoc = function(req, res){};
```

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

5. We can now expand the **`putnewdoc()`** function to interrogate the REST client's body values for name and weight. **NOTE: body parser must be setup properly for this to work:**

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

6. Create a variable and point it to the **`Weight`** object, which represents our database

```
    exports.putnewdoc = function(req,res){
        let empName = req.body.empName;
        let empWeight = req.body.empWeight;
        const weight = new Weight();
```

7. Use the new variable and its properties to pass values from the form to the database properties

```
        const weight = new Weight();
        weight.empName = empName;
        weight.empWeight = empWeight;
```

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

```
exports.putnewdoc = function(req,res){
      let empName = req.body.empName;
      let empWeight = req.body.empWeight;
      const weight = new Weight();
      weight.empName = empName;
      weight.empWeight = empWeight;
      weight.save({}, function(err) {
            if (err)
                  res.end(err);
            res.end(`Created ${empName}`);
  });
};
```

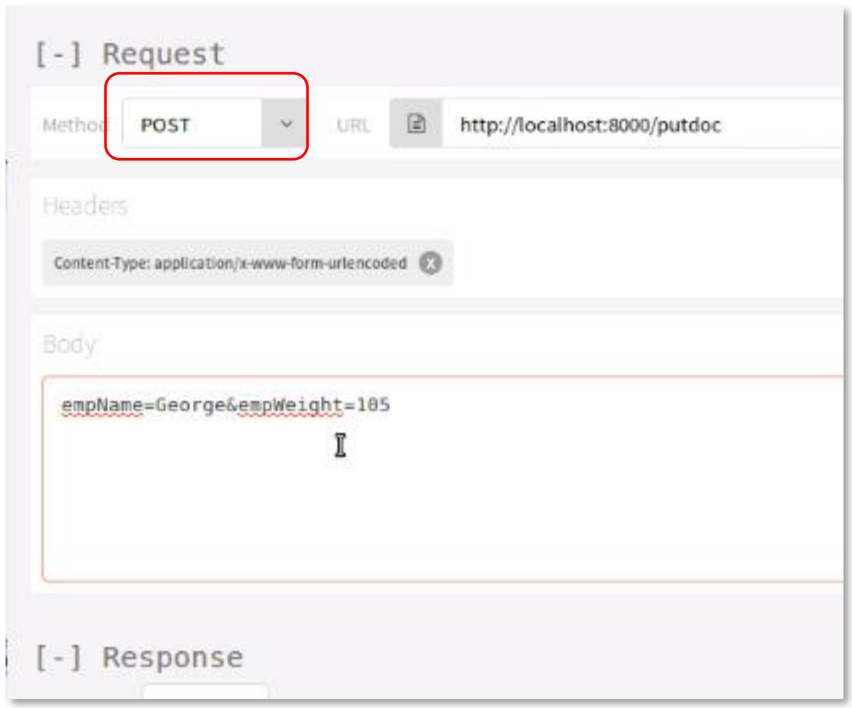
Test the new function using the **REST** client.

**NOTE:**

**1. body parser must be installed properly for this to work**

**2. CORS must be enable in the browser**

**3. body-parser must be installed and *app.use(bodyparser)* must be directly underneath *const app = express();***


[-] Request
Method
POST
URL
http://localhost:8000/putdoc
Headers
Content-Type: application/x-www-form-urlencoded
Body
[-] Response

```
empName=George&empWeight=105
```

## PART 13 – EXPANDING THE CONTROLLER TO UPDATE A DOCUMENT IN THE DATABASE

1. Copy the folder called **`Part11`** paste it inside of **`Day05`** and then rename it to **`part12`**.

2. Open a terminal inside of **`Part12`** and run the command **`npm install`** which will install everything that **`part11`** had, it will use the json file from that directory.

3. In the routes.js file, copy any of the previous route lines and change the route to be **`updatedoc`**.

```
app.route('/updatedoc).put(controller. updatedoc);
```

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

4. Create a matching function in the controller.js file, in fact we can just copy, paste and edit the **`putnewdoc()`** function.

```
        exports.updatedoc= function(req,res){};
```

   In this function, we get the name and weight from an HTML form, NOT the url.

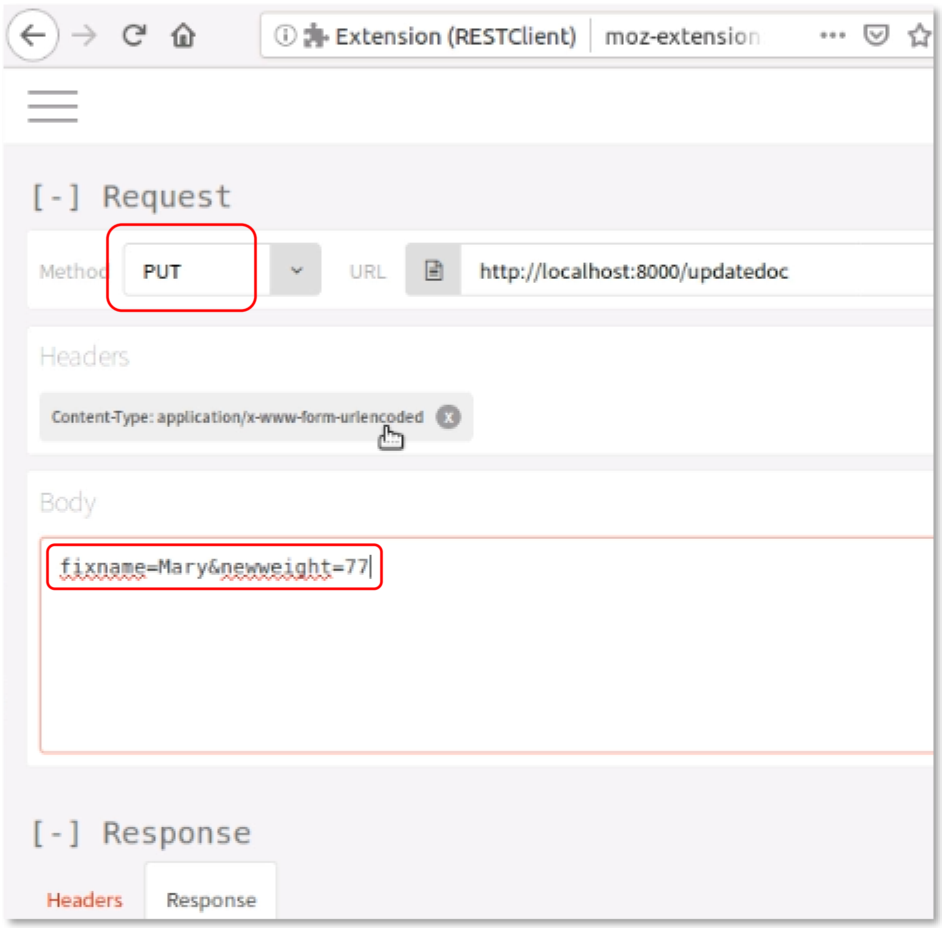
5. Since we did most of what is needed in the function in the **`putdoc()`** function, just copy paste and change

```
exports.updatedoc = function(req, res) {
            let fixName = req.body.fixname;
            let newWeight = req.body.newweight;
            var query = { empName : fixName };
            var data = { $set : {empWeight : newWeight } }

  Weight.updateOne(query, data, function(err, result) {
    if (err)
      res.send(err);
    res.end(`Updated ${fixName}`);
  });
};
```

   In this function, we use the **`updateOne()`** method of the weight object. Also we need to first find the record we need to update and then update it by using the **`$set`** keyword.

   Test the new function using the REST client. NOTE: body parser must be installed properly for this to work.



PUT

http://localhost:8000/updatedoc

Content-Type: application/x-www-form-urlencoded

```
fixname=Mary&newweight=77
```

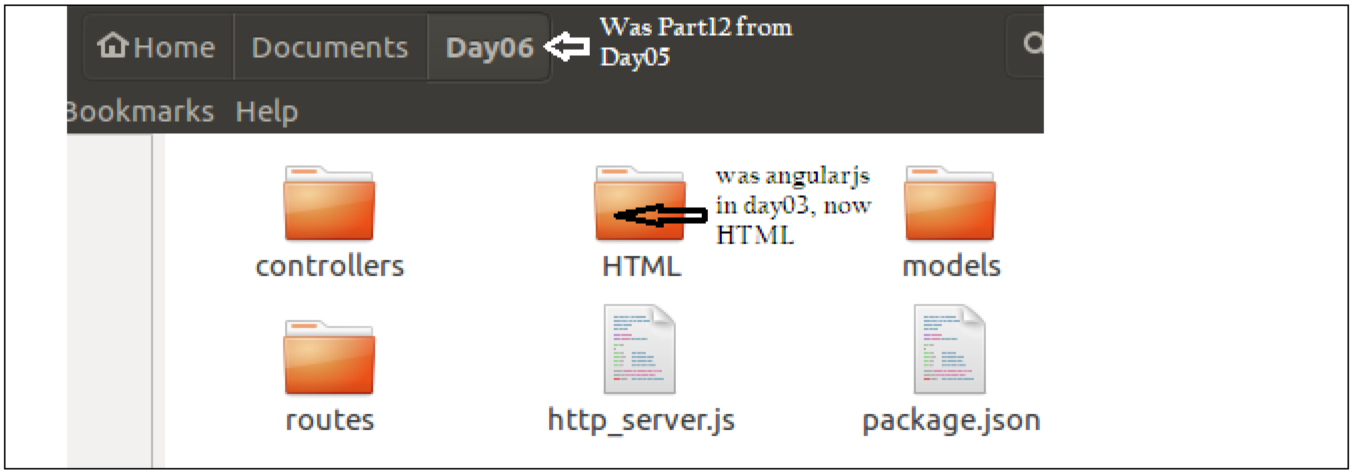
# Day06 Deployment

## Part 1 – Folder Setup

1. Copy the folder called **`Part12`** from **`Day05`** paste it where all the other days are and rename it to **`Day06`**.

2. Copy the folder called **`AngularJS`** from **`Day03`** paste it inside of the folder you just renamed to **`Day06`**. Rename **`AngularJS`** inside of **`Day06`** to just **`HTML`**. It should contain all the HTML/JS/CSS from the third day of sessions.

3. Your **`Day06`** folder should look like the image below, before running **`npm install`**.



4. Run **`npm install`** to create the **`node_modules`** folder and setup the environment.

5. First task is to serve the **`HTML`** page we built on **`Day03`**. Now that we have a **`controller`** file, everything that goes to the client will pass through there and the root function will handle serving our HTML page when users land on our root which at the moment is http://localhost/8000. We would need to change line 3 of controller.js to server the index.html file instead.

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

6. Execute **`node http_server`** and go to the address from #5. 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/models');
    exports.getdefault=function(req, res){
        res.sendFile('../HTML/index.html');
    };
```

Remember to stop and start the service

7. 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/models');
    const path = require("path");
    exports.getdefault=function(req, res){
```

8. 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'));
    };
```

9. 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. Open routes.js and include the following lines:

```
    module.exports = function(app){
        const express= require('express');
        app.use(express.static(__dirname + '/../HTML'));
        let controller = require('../controllers/controller');
```

Although we had to require **`Express`** in the http_server.js file, we still have to do it again in this file. Also we just need the static method to know where the directory is that contains our html/css/js files.

10. At this point, if you navigate to team weights in the navigation menu, you should see some records there already. If you do not see anything then you would need to complete the teamweights.html file.

11. Open the scripts.js file inside of the scripts folder. Make sure that the AngularJS controller is obtaining its data from the URL and not the JSON text file.

```
let file = "json.txt";
let url = "http://localhost:8000/getallrecords";
//
let app = angular.module('SkillsApp', [] );
app.controller('Weights', function($scope, $http) {
    $http.get(url).then(function(response){
            $scope.allWeights = response.data;
```

12. There may be some left over problems from previous days and parts, so hit the f12 key to see if there are any errors and fix them. For example on the home page, there may be a message that angular is not defined. Just include the AngularCDN in the head of the document. Do this for any page that throws this error.

## 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, should be around line 29. Add a new pair of **div** tags underneath in order to wrap the **ng-repeat** block of code.

```
<div id="showRecords">
    <div id="records" ng-repeat="emp in allWeights">
            {{emp.empName}} weighed in at {{emp.empWeight}} Kgs
    </div>
</div>
```

Give it an id as well.

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

```
#showRecords {
  margin-left:34px;
  width:80%;
}
```

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

3. 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:

```
#showRecords div:nth-child(odd) {
    background: lightgray;
    display: block;
}
```

4. (optional)
   you may want to include row numbers, and we can tap into Angular's **`$index`** for that.

```
<div id="records" ng-repeat="emp in allWeights">
    {{$index + 1}} {{emp.empName}} weighed in at {{emp.empWeight}} Kgs
</div>
```

5. If you did this, you may notice that the start of each sentence does not really line up. One trick is to introduce another powerful Angular feature called **`ng-if`**. Change the code to the following:

```
<div id="showRecords">
    <div id="records" ng-repeat="emp in allWeights">
            <span ng-If="($index + 1)<10">&nbsp;&nbsp;{{$index + 1}} {{emp.empName}}
weighed in at {{emp.empWeight}} Kgs</span>
            <span ng-If="($index + 1)>9">{{$index + 1}} {{emp.empName}} weighed in at
{{emp.empWeight}} Kgs</span>
    </div>
</div>
```

   This can be improved using ng templates and the **`ng if else`** structure.

## PART 3 – CONFIGURING MYWEIGHTS

For myweights.html, we already have the file created and almost ready to go. We would have to create an html element to display the individual's record, once found. I am building this on the assumption that we are displaying one record, but in reality, we could be displaying several records, so in that case you can follow the code for teamweights.html.

1. Remove all the **`p`** tags except for one to show the database record and give it an id:

```
            </div>
    </form>
    <p id="displaySingle">On [date] you weighed [empWeight] Kgs.</p>
</div>
<div id="aside">
```

2. In the controller.js file, copy the **`getbyname()`** function and rename it something like **`getbyformname()`**.

```
    exports.getbyformname=function(req, response){
      let empToFind = req.params.employeeName;
        Weight.find({empName:empToFind}, function(err, results){
                if (err)
                    response.end(err);
                response.json(results[0].empWeight);
        });
    };
```

3. The values wont be coming via the url, so **`params`** wont work, we need to use **`body`**. Also in the form, our field is called **`empName`** not **`byname`**, so change this also.

```
    exports.getbyformname = function(req, res) {
     Weight.find({empName:req.body.empName}, function(err, results) {
       if (err)
         res.send(err);
       res.json(results);
     });
    };
```

4. Create a route to point to this function, make sure it's a **`post`** request/method.

```
        app.route('/getbyformname').post(controller.getbyformname);
```

5. We would need to create a new function in the scrpts.js file so that we can post to the proper **`api`**, right now we have a **`get`** and a **`frmSubmit()`**. Create a new function within the **`controller`** scope, just copy the **`frmSubmit()`** function and modify the name first.

```
        $scope.frmFindSingle = function(){
                $http({
                        method  : 'POST',
                        url     : 'http://localhost:8000/getbyformname',
                        headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
                        data    : 'empName='+$scope.empName + '&empWeight='+$scope.empWeight
                });
        };
```

6. Also remove the e**`mpWeight`** part from the **`data`** line, this is what we are trying to find.

```
        $scope.frmFindSingle = function(){
                $http({
                        method  : 'POST',
                        url     : 'http://localhost:8000/getbyformname',
                        headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
                        data    : 'empName='+$scope.empName
                });
        };
```

7. The controller method in controler.js will return whatever the mongo `find()` method gathers, which is quite a lot, so lets reconfigure our new `frmFindSingle()` method to pick out the data we need:

```
                    $scope.frmFindSingle = function(){
                    $http({
                            method  : 'POST',
                            url     : 'http://localhost:8000/getbyformname',
                            headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
                            data    : 'empName='+$scope.empName
                    }).then();
                    };
```

Because angular's `http()` method works on the basis of `promises` we can chain a `then()` method onto it, to handle any response from the `http()` method.

8. Within the `then()` method we can place an anonymous function to handle the response:

```
            $scope.frmFindSingle = function(){
                    $http({
                            method  : 'POST',
                            url     : 'http://localhost:8000/getbyformname',
                            headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
                            data    : 'empName='+$scope.empName
                    }).then(function(response){});
            };
```

The `response` object will be passed into the function, once it is chained in this way

9. With the function in place, lets interrogate the `response` object to get the weight from the document we just found. Pass this to Angular's `$scope` object.

```
            $scope.frmFindSingle = function(){
                    $http({
                            method  : 'POST',
                            url     : 'http://localhost:8000/getbyformname',
                            headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
                            data    : 'empName='+$scope.empName
                    }).then(function(response){
                            $scope.empWeight = response.data[0].empWeight;
                    });
            };
```

10. With our value stored, we just need a way to display it on the HTML file itself, so we can use the sentence we already have:

```
            </form>
            <p id="displaySingle">{{empName}}, on [date] you weighed {{empWeight}} Kgs.</p>
        </div>
```

If the myweights.html file was not setup like the teamweights.html file, this may not work

11. We now have to make sure that angular is setup properly, so in the **`maincontainer`** div tag of the html, add the **`ng-app`** directive:

```
        </ul>
    </nav>
    <div id="maincontainer" ng-app="SkillsApp">
        <div id="maincontent">
```

12. In the same way, add the directive **`ng-controller="Weights"`** to the **`maincontent`** div tag underneath

```
        </ul>
    </nav>
    <div id="maincontainer" ng-app="SkillsApp">
        <div id="maincontent" ng-controller="Weights">
                <h2>My Records</h2>
                <form>
```

13. Make sure you call the correct function from the **`form`** tag

```
    <div id="maincontent" ng-controller="Weights">
        <h2>My Records</h2>
        <form name="frmCollectWeights" ng-submit="frmFindSingle()">
                <div>
                        <label for="empName">Name</label>
```

14. Make sure that the form input boxes have the proper Angular directives

```
    <form name="frmCollectWeights" ng-submit="frmFindSingle()">
        <div>
                <label for="empName">Name</label>
                <input id="empName" type="text" ng-model="empName"/>
        </div>
        <div>
```

15. Stop and start the server, then test the new route and method

## PART 4 – ADDING A NEW RECORD

The HTML file enterweight.html is being used to add a new record to the database. We have to configure that file to work with the **`putnewdoc`** endpoint.

1. Open the scripts.js file and make sure that the **`url`** has the proper value, it should be pointing to **`http://localhost:8000/putdoc`**

2. Remove the `console.log` line if there is one, here is the entire function for `frmSubmit()`

```
$scope.frmSubmit = function(){
    $http({
            method  : 'POST',
            url     : 'http://localhost:8000/putdoc',
            headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
            data    : 'empName='+$scope.empName + '&empWeight='+$scope.empWeight
    });
};
```

3. In a case like this, it may be a good idea to send our user to a totally different page so that they don't try to enter the same record again. We can use the `.then()` function for this.

```
    $scope.frmSubmit = function(){
      $http({
        method  : 'POST',
        url     : 'http://localhost:8000/putdoc',
        headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
        data    : 'empName='+$scope.empName + '&empWeight='+$scope.empWeight
      }).then(function(response){
            if(response.data != null){
                window.location = "creatednewdoc.html";
            };
     });
   };
    //
```

4. For createnewdoc.html, just copy one of the AngularJS files like teamweights.html and remove the repeating code. Replace that code with something like below.

```
</nav>
<div id="maincontainer">
    <div id="maincontent" ng-controller="Weights">
            <h3>New document created</h3>
            <p>Would you like to create another record?</p>
            <p><button ng-click="gotoEdit()">Yes</button></p>
    </div>
    <div id="aside">
            <div id="section">
```

So basically, remove everything between the `maincontent` div and replace those lines with the ones highlighted

5. In the scripts.js file, create a new function within the `Weights` scope called `gotoEdit()` and use that function to send the user back to enterweight.html

```
                    $scope.empWeight = response.data[0].empWeight;
            });
    };
//
    $scope.gotoEdit = function(){
            window.location = "enterdata.html";
   }
    //
});
//
```

Test the application

## PART 5 – DELETING A RECORD (ADMIN FOLDER)

Deleting a document should be an admin function and should be in a separate folder. We also have HTML files that do some of the work of sending off a name to be deleted, just like we can find a name.

1. Create a new folder called admin inside of HTML and copy the myweights.html file into that folder and rename it to deletedoc.html

2. Since deletedoc.html is within a folder one level deep, we may need to adjust the paths to reflect this. So wherever we are referring to files on the local file system, we need to put **../** in front of what we have already:

```
    <div class="wrapper">
        <header>
                <img id="logo" src="../images/chart.gif" />
                <h1><a href="../index.html">Skillsoft Weight Tracker</a></h1>
                <meta charset="utf-8"/>
                <link rel="stylesheet" type="text/css" href="../styles/styles.css" />
        </header>
    .
    .
    .
                </div>
        </div><!--closes the container div-->
        <script src="../scripts/scripts.js"></script>
    </body>
```

   Also the file is located in the admin folder so:
   **`http://localhost:8000/admin/deletedoc.html`**

3. Change the page title, button and message to reflect that this is a delete. We would also create a new method to handle interaction with the API method

```
    <div id="maincontent" ng-controller="Weights">
        <h2>Deleting A Record</h2>
        <form name="frmCollectWeights" ng-submit="frmDeleteSingle()">
                <div>
                        <label for="empName">Name</label>
                        <input id="empName" type="text" ng-model="empName"/>
                </div>
                <div>
                        <button>Delete Record</button>
                </div>
        </form>
        <p id="deleteMessage">{{empName}}, was deleted.</p>
    </div>
    <div id="aside">
```

4. Turn attention now to the controller.js file and we already have a delete function that uses the `URL` params object, we can copy this function and configure it to use the `body` instead.

```
    exports.deletebyformname = function(req, res) {
      let delName = req.body.empName;
      Weight.deleteOne({empName:delName}, function(err, result) {
        if (err)
          res.send(err);
        res.end(`Deleted ${delName}`);
      });
    };
```

Everything else should work as is

5. Create a corresponding route in routes.js

```
        app.route('/getbyname/:byname').get(controller.getbyname);
        app.route('/deletebyname/:byname').delete(controller.deletebyname);
        app.route('/deletebyformname').delete(controller.deletebyformname);
        app.route('/putdoc').post(controller.putnewdoc);
        app.route('/updatedoc').put(controller.updatedoc);
```

6. In scripts.js we need to create a function called `frmDeleteDoc()` as we eluded to in #3 above. Copy the `frmSubmit()` function and change it to interact with the corresponding `API`.

```
    $scope.frmDeleteDoc = function(){
        $http({
                method  : 'DELETE',
                url     : 'http://localhost:8000/deletebyformname',
                headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
                data    : 'empName='+$scope.empName
        }).then(function(response){

        });
    };
```

7. According to the API from #4 above, we are expected to get back some kind of message. We can either pass this message to our HTML or use Angular's features to hide a prepopulated message. This has to be done in 2 places, in the deletedoc.html, add the `ng-hide` directive to the `deleteMessage div` tag

```
        <div>
                <button>Delete Document</button>
        </div>
        <div ng-hide="deleteMessage">
                <p id="deleteMessage">{{empName}}, was deleted.</p>
        </div>
    </form>
```

Then in scripts.js, complete the `then()` function to show the message on getting the proper response from the `API`

```
        data    : 'empName='+$scope.empName
    }).then(function(response){
        if(response.data != null){
                $scope.deleteMessage = false;
        };
    });
```

Also still in scripts.js, you would need to hide the **message div** prior to executing the call to the **API**. This happens at the very top of the file, but inside of the controller's scope.

```
    let app = angular.module('SkillsApp', [] );
    app.controller('Weights', function($scope, $http) {
        $scope.deleteMessage = true;
        $http.get(url).then(function(response){
                $scope.allWeights = response.data;
        });
```

8. Prior to testing the HTML form, it would be a good idea to test the API using the **REST** client.

Here is the complete **frmDeleteDoc()** function

```
        $scope.frmDeleteDoc = function(){
          $http({
            method  : 'DELETE',
            url     : 'http://localhost:8000/deletebyformname',
            headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
            data    : 'empName='+$scope.empName
          }).then(function(response){
               if(response.data != null){
                   $scope.deleteMessage=false;
               };
         });
      };
```

Here is the controller.js **API**

```
    exports.deletebyformname=function(req, response){
        let delName = req.body.empName;
        Weight.deleteOne({empName:delName}, function(err, results){
                if (err)
                   response.send(err);
                response.end(`Deleted ${delName}`);
        });
    };
```

## PART 6 – EDITING A RECORD (ADMIN FOLDER)

Edits are not as easy as the other **CRUD** operations. We first have to find the record (myweights.html) then display the document's data inside of two **input** tags (enterweight.html), then finally edit the record and send it back to the server.

Based on the pattern so far, we would need to use an existing **HTML** file and change the **HTML** code, create a new function in scripts.js and make sure we have a corresponding **API** to hook into.

1. Since we will be using both **input** tags, we can copy enterweight.html, paste and rename to editweight.html. This should go into the **admin** folder also.

2. Fix the paths as we did for deleting a record.

3. We should hide the weight input box initially and then show it if a record was found. As the **input** and **label** is already wrapped in a div tag (**formSeparator**) we can apply **ng-hide** to the entire **div** and hide both elements

```
        <input id="empName" type="text" ng-model="empName"/>
    </div>
    <div class="formSeparator" ng-hide="hideWeight">
        <label for="empWeight">Your Weight Today</label>
        <input id="empWeight" type="text" ng-model="empWeight"/>
    </div>
    <div>
```

4. Change the text on all elements appropriately

```
    <div id="maincontent"  ng-controller="Weights">
        <h2>Editing weights</h2>
        <form name="frmCollectWeights" ng-submit="frmSubmit()">
                <div>
                        <label for="empName">Name to Edit</label>
                        <input id="empName" type="text" ng-model="empName"/>
                </div>
                <div class="formSeparator" ng-hide="hideMessage">
                        <label for="empWeight"> Current Weight</</label>
                        <input id="empWeight" type="text" ng-model="empWeight"/>
                </div>
                <div>
                        <button>{{ editingButton}}</button>
                </div>
        </form>
```

Notice the button text will change based on *find* or *edit*.

Here is all the code so far in the form area:

```
<form name="frmCollectWeights"  ng-submit="frmEditDocument()">
    <div>
            <label for="empName">Name to Edit</label>
            <input id="empName" type="text" ng-model="empName" />
    </div>
    <div class="formSeparator" ng-hide="hideWeight">
            <label for="empWeight">Current Weight</label>
            <input id="empWeight" type="text"  ng-model="empWeight" />
    </div>
    <div>
            <button>{{editingButton}}</button>
    </div>
</form>
```

5. In scripts.js we need to declare the `hideWeight` and `editingButton` variables and assign the appropriate values. Do this at the top of the `controller` function.

```
let app = angular.module('SkillsApp', [] );
//
app.controller('Weights', function($scope, $http) {
    $scope.hideWeight=true;
    $scope.editingButton="Find Document";
    //
    $http.get(url).then(function(response){
```

6. In scripts.js we need to create a new function, lets call it `frmEditDoc()`, so this is the function that our `HTML` form will call also. **Make sure that this is changed in editdocument.html.**

```
    $scope.frmEditDoc = function(){
            $scope.frmFindSingle();
    }
```

7. If `frmFindSingle()` does find the record, we will know because `$scope.empWeight` will contain a value greater than zero.

8. In order to have access to that value, we would need to call back into the `frmEditDocument()` function. It means we need to first change the function to accept a function parameter and postback using a chained `then()` method:

```
$scope.frmFindSingle = function(callback){
    $http({
            method  : 'POST',
            url     : 'http://localhost:8000/getbyformname',
            headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
            data    : 'empName='+$scope.empName
    }).then(function(response){
            $scope.empWeight = response.data[0].empWeight;
    }).then(callback);
};
```

9. When we call **`frmFindSingle()`** we need to wire up a call back function. We now have access to the value in **`empWeight`**. Lets use that to first reveal the **`empWeight`** input box and insert into that box the value returned from **`frmFindSingle()`**.

```
        $scope.frmEditDocument = function (){
                $scope.frmFindSingle(function(){
                        if($scope.empWeight  != ""){
                                $scope.hideWeight = false;
                        }
                });
        }
```

Because **`empWeight`** input box is wired up, it gets the weight value automatically

10. Once this works, we need to change the **`button`** to display something like "Update Document", because we have already found the document and its corresponding weight. Do this in the same function call

```
        $scope.frmEditDocument = function (){
                $scope.frmFindSingle(function(){
                        if($scope.empWeight  != ""){
                                $scope.hideWeight = false;
                                $scope.editingButton = "Update Document";
                        }
                });
        }
```

As the form is loaded, "Find Document" will be displayed on the button, but this will have to change to "Update Document" once the record is found.

11. Initially the button will have the text "Find Document", but if the document is found, then that button's text changes to "Update Document". We need to wrap up our code into an **`if`** statement to test for this text.

```
        $scope.frmEditDocument = function (){
                if($scope.editingButton !== "Update Document"){
                        $scope.frmFindSingle(function(){
                                if($scope.empWeight  != ""){
                                        $scope.hideWeight = false;
                                        $scope.editingButton = "Update Document";
                                }
                        });
                } else {
                        $scope.frmDoEdit();
                }
        }
    //
```

We now have to write the **`frmDoEdit()`** function. We can't use the original **`frmSubmit()`** because that works with brand new records and edits are usually associated with the **`HTTP PUT`** verb

12. The **`frmDoEDit()`** function

```
                $scope.frmDoEdit = function(){
                $http({
                        method  : 'PUT',
                        url     : 'http://localhost:8000/updatedoc',
                        headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
                        data    : 'fixName='+$scope.empName + '&newWeight='+$scope.empWeight
                });
        }
```

Notice the method value

13. If you look at the controller.js file, and the **`updatedoc()`** method, you will see that it is returning a message upon a successful edit, we can use that message in our front end. Create a new variable and a new **`div`** underneath the **`button div`**, we will reveal the message from the database there:

```
                                <div>
                <button>{{editingButton}}</button>
        </div>
        <div id="postMessage">
                        {{afterEdit}}
        </div>
    </form>
```

14. Back to the scripts.js file, chain the **`then()`** method to the frmDoEdit() method. Use the variable **`afterEdit`** so that the message appears in the HTML.

```
        $scope.frmDoEdit = function(){
                $http({
                        method  : 'PUT',
                        url     : 'http://localhost:8000/updatedoc',
                        headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
                        data    : 'fixName='+$scope.empName + '&newWeight='+$scope.empWeight
                }) .then(function(response){
                        if(response.data != "")
                                $scope.afterEdit = response.data;
                        else
                                $scope.afterEdit = "";
         });
        }
```

Here is the entire frmDoEdit() function:

```
    $scope.frmDoEdit = function(){
        $http({
                method  : 'PUT',
                url     : 'http://localhost:8000/updatedoc',
                headers : {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8'},
                data    : 'fixName='+$scope.empName + '&newWeight='+$scope.empWeight
        }).then(function(response){
                if(response.data != "")
                        $scope.afterEdit = response.data;
                else
                        $scope.afterEdit = "";
        });
    }
```

15. We could improve the security of frmEditDocumen() by first checking that the value in empWeight is in fact a number, if not we should display an error:

```
        $scope.frmEditDocument = function (){
                if($scope.editingButton !== "Update Document"){
                        $scope.frmFindSingle(function(){
                                if( weight = parseFloat($scope.empWeight )){
                                        if(weight > 0 ){
                                                $scope.hideWeight = false;
                                                $scope.editingButton = "Update Document";
                                        } else {
                                                $scope.afterEdit = "An error occured";
                                        }
                                }
                        });
                } else {
                        $scope.frmDoEdit();
                }
        }
```

Notes.

1. It may be better to send the user to a totally different page like we did after a new document was inserted. You could reconfigure the `gotoEdit()` function to do this.
2. After the document is updated, there is some extra data showing up, for example result 1. This is coming from the API, so either adjust it there or remove this last part, but if you are sending them to a new page, then there is no point.

# Appendix

## Appendix A - CORS

The same-origin policy is a security model for web applications. Under the policy, a web browser permits scripts contained in one web page to access data in a different web page, as long as, both web pages have the same origin.

An origin is usually a combination of URI, host, and port number. This policy restricts a malicious script on one page from accessing and changing sensitive data on another web page via that page's DOM.

When a web application makes a requests from a resource that has a different origin (domain, protocol, and port) than its own, this is known as a **cross-origin HTTP request**.

Cross-Origin Resource Sharing (CORS) is a W3C specification that allows HTTP headers to permit a browser to allow a web application running at one origin or domain to access resources from a server at some other web server origin.

An example of a cross-origin request:

A web page's JavaScript code served from http://localhost uses **XMLHttpRequest** to make a request for resources on http://abc.domain.com/

For security reasons, browsers restrict cross-origin HTTP requests coming from scripts. For example, XMLHttpRequest and the Fetch API. This means that a web application using those APIs can only request HTTP resources from the same server the application was loaded from.

If the response from the other origin includes the right CORS headers, as in when we install a CORs plugin, the request succeeds.

In this application, we need to install a CORS plugin for the browser you are using.

For Chrome, there is a plugin located here:
https://mybrowseraddon.com/access-control-allow-origin.html

For Firefox, you may use the one shown on the Live Learning session:
https://addons.mozilla.org/en-US/firefox/addon/cors-everywhere/

## Appendix B – Editors

The application is keeping close to original/raw code as possible, so we are staying away from Integrated Development Environments. However this does not mean that these are not productive or useful, however we are still in a teaching environment.

We are mainly using Notepad++ but this may change in the future. Installation directions for most major operating systems may be found on their website: https://notepad-plus-plus.org/

Other good editors include Atom, Sublime Text and Brackets.

## Appendix C – REST Client

Restful APIs allow clients such as browsers to perform CRUD operations on some data store. This is potentially dangerous and usually browsers will only be able to access resources like web pages or images.

In order to perform full CRUD operations, the browser needs some help. This is where a REST client comes in handy.

A REST client will allow the developer to make those edits, deletes and updates directly from the browser, without implementing the full web application.

The one used in the application for Mozilla Firefox is located here:

https://addons.mozilla.org/en-US/firefox/addon/restclient/

Chrome users should install Postman, which is now a stand-alone application:
https://www.getpostman.com/

If you use any other browser, please search for a REST client that is compatible with your browser.