

WEB 322 Assignment 1: Report

Newman Law  
134543198  
June 19, 2020  
Professor Kadeem Best

## TABLE OF CONTENTS

<b>INTRODUCTION</b>	<b>3</b>
<b>JAVASCRIPT</b>	<b>4</b>
<b>NODE JS</b>	<b>4</b>
<b>NPM</b>	<b>5</b>
<b>EXPRESS</b>	<b>6</b>
<b>EXPRESS-HANDLEBARS</b>	<b>8</b>
<b>MODULES</b>	<b>10</b>
<b>HTML</b>	<b>11</b>
<b>MAIN TEMPLATE</b>	<b>13</b>
<b>HOME/LISTINGS PAGE</b>	<b>13</b>
<b>LOGIN/REGISTRATION PAGE</b>	<b>13</b>
<b>CSS</b>	<b>15</b>
<b>MAIN TEMPLATE</b>	<b>15</b>
<b>LISTINGS/HOME CSS</b>	<b>16</b>
<b>LOGIN/REGISTRATION CSS</b>	<b>16</b>

**Introduction**

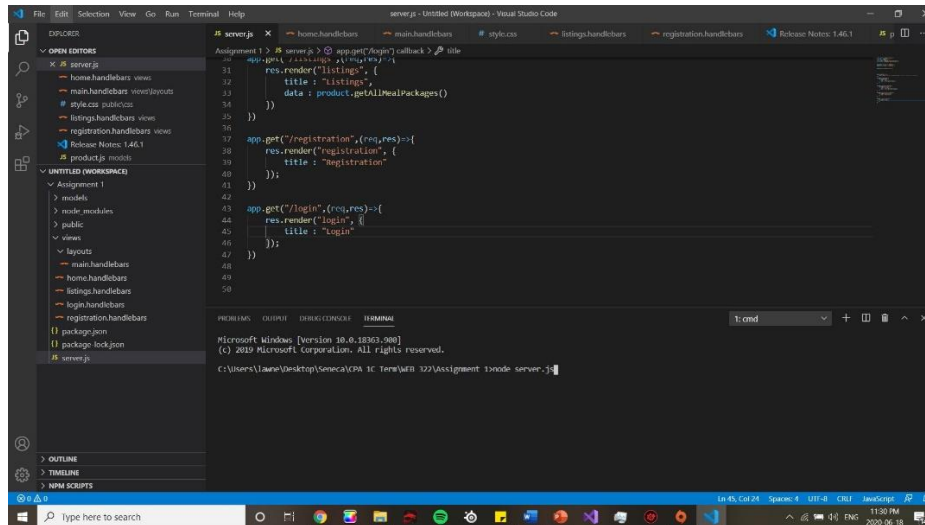
In this assignment, I was given the task of creating a website as a Full Stack developer, specifically a website for Live Fit Food. Using Visual Studio Code, I had to have a thorough understanding of HTML5, CSS3, JavaScript, “Node.JS”, Express, and Handlebars. For this assignment, I will be separating the content into three different sections: JavaScript, HTML5, and CSS3. Each of these sections build off on each other, both in the report as well as in the code.

## **JavaScript**

Before this assignment was introduced, my concept of JavaScript only ranged from a front-end perspective. The most knowledge I had was node JS, as well as understanding the DOM tree, arrays, object literal, and the use of var. Throughout this term, so far, I have learned to implement ES6, as well as using different JavaScript frameworks in order to understand how to create web sites from a back-end perspective. JavaScript, itself, is a programming language that is considered high-level, as well as a loosely typed language. Loosely typed, meaning that the interpreter itself does not require the programmer to define what the data type the variable is when implemented. For instance, in C++, I would use int, char, etc. to define the data type. In JavaScript, the language allows for me to just define the variable as either "let" or "const". JavaScript is also considered a high-level language, as it displays strong abstraction from the details of the computer. This allows for the language to be easier to read, write, and maintain.

## **Node**

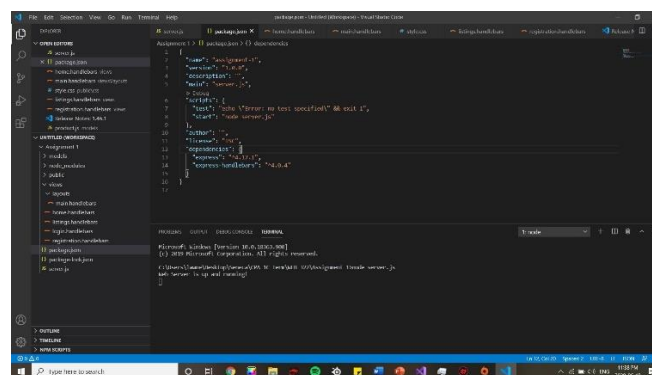
Node was extremely useful when running JavaScript code, both for debugging certain functions without the need of a JavaScript engine provided by a web browser, as well as running my Web Server. Node itself is a JavaScript runtime that allows programmers to run JavaScript outside of a browser. It also allows additional features to JavaScript beyond the traditional language itself and allows asynchronous programming. I installed Node with the following link (<https://nodejs.org/en/download/>) and have used node throughout this project in order to run my server, as well as test JavaScript code. To use node, I simply opened a new terminal in Visual Studio Code and typed in the following command: node (filename).



**Figure 1:** Demonstrating use of node js. Observe the new terminal created, as well as the syntax with node server.js, server.js being the name of the file.

## NPM

By downloading and implementing node in our machine, we can use a manager known as NPM, or Node Package Manager. This Manager is an official package manager for the Node.js and is automatically bundled and installed automatically with the environment. With this knowledge, I used NPM first to download a package.json file. This was initiated by typing in the terminal “npm init”, which would have a number of questions that the user would have to answer, or “npm init -y”, which would automatically answer the questions for the user. This then creates a package.json file for your folder, as well as automatically identifying what is the entry point file for your web server (in the case this assignment, it is server.js).



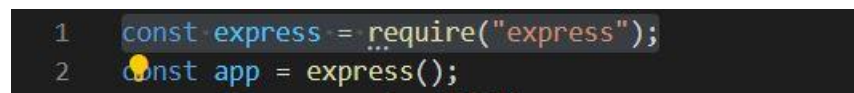
**Figure 2:** Example of a package.json file created by npm init -y. NOTE: this was created already with installation of a 3<sup>rd</sup> party packages (express and express-handlebars)

In the figure above, you can observe how there are dependencies in the package.json file. “Express” and “Express-handlebars” do not come when you initiate the command “npm init” or “npm init-y”, as it is a third-party installation. An easy understanding of a package.json file is thinking of it as a manifest for applications, modules, packages, etc.

### **Express:**

Express is as a JavaScript framework that is used in building web applications and APIs. It allows for handling request and routes easier when compared to vanilla Node.js; therefore, making it a standard for most back-end web developers. to install express, first create a package.json with npm init (-y) and then type into the terminal: npm i express. Express is a middleware, which accepts request from a HTTP as well as sending out response objects. In other words, express has the power to access request and response objects, and allows for manipulation and change for both objects as well. In my server.js code, we can see examples of express through code such as **const express = require(“express”), const app = express(), app.use(express.static(‘public’)), app.listen, and app.get.**

The first code, const express = require(“express”), as well as the second code, const app=express(), allows for express to be used in your JavaScript code. The first code imports the express module. The second code allows for me to create object called app, which will then have methods for routing HTTP requests, configuring the middleware, rendering HTML views, as well as registering a template engine (will be described more in express-handlebars section of report).

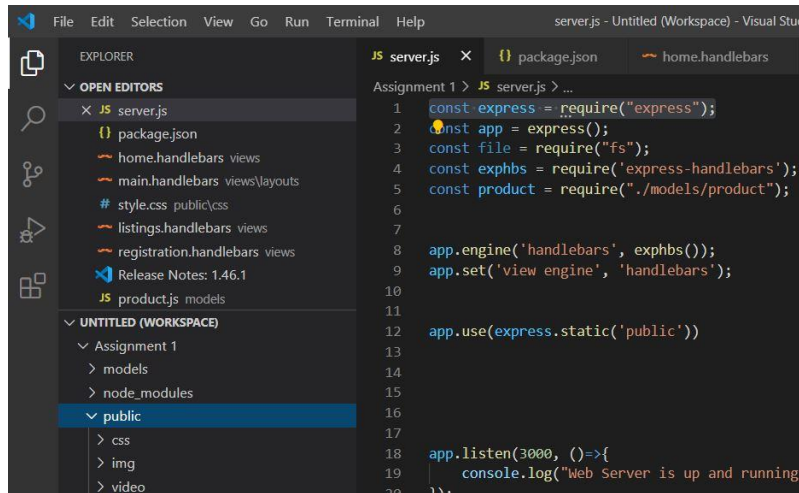


```
1 const express = require("express");
2 const app = express();
```

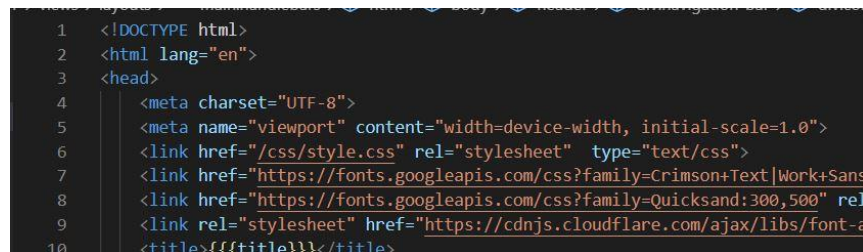
**Figure 3:** Example of implementing require(“express”) and our object for express. NOTE: the object can be named whatever the programmer desires, with “app” being best practice.

The third code, known as express.static(‘folder’), enables express to use static files. To explain this concept, if we do not include static, we must create a route for every single file that is static. This is because express cannot understand file path names; therefore, when my browser sends an HTTP request based on a file path name in my HTML (for instance, css/style.css),

express won't be able to understand it, as it only understands routes. By implementing `express.static`, whatever is inside the bracket will serve as the root directory from which to serve static assets. The function determines the files to serve by combining the `req.url` with the provided root directory. When the file is not found in the root directory, it will call `next()` instead of sending a 404 response. In the case of my code, I stored all our static assets in a folder called `public`.



**Figure 4:** Example of using `express.static` to access static files in my directory named “public”



**Figure 5:** href must use file path name of “/css/style.css” due to public being the root directory

Finally, `app.listen` starts our server and listens on a specific port for connections. In the case of my assignment, I used port 3000 and displayed a message in my terminal if the Web Server is running. Furthermore, `app.get` is an example of a GET method route and is derived from one of the HTTP methods. A GET request to a server only occurs when a user clicks on a hyperlink or types a URL. The GET request will ask the server to give a resource/response, with express acting as a middleware. In our express function, `get()`, the first parameter matches whatever route is being called. From the specific route, the server will then send a response based on what is inside the specific `app.get` function. To put simply, the `get` function is just a

callback function. The second parameter in our `app.get (req, res)` are two objects. `Req` is an object containing information about the HTTP request that raised the event. In response to `req`, the `res` sends back the desired HTTP response. In the case of our figure below, `res.render` sends an HTTP response using `express-handlebars`.

```
app.listen(3000, ()=>{
  console.log("Web Server is up and running!") //will let us know if it even works
});

app.get("/",(req,res)=>{
  res.render("home", {
    title : "Home",
    data : product.getAllProducts()
  })
})
```

**Figure 6:** Displaying `app.listen` and `app.get` and its utilization in my code.

### Express-Handlebars:

Express-Handlebars is a templating engine for express. To install `express-handlebars`, I opened a new terminal and typed in the command: `npm i handlebars`. A template engine allows programmers to create template files and a main layout file in your application. In my `main.handlebars` (the main layout file), I was able to use a layout that would be the same throughout all my different routes. The only difference in the routes was the data in each page. This allowed for dry code, as I did not have to repeat the same code over and over again.

After installation of `handlebars`, I would have to use the following codes to utilize express: `const expHbs = require('express-handlebars'), app.engine('handlebars', expHbs()), app.set('view engine', 'handlebars'),` and `res.render` inside our `app.get` function.

The first three codes allows for `handlebars` to work in our JavaScript. `Res.render` allows for me to access different template files. This is a form of dynamic injection, as I can implement different templates files data into my `main.handlebars`, or known as our main layout file.

After these codes are written down, I had to create a folder named `views`, with another folder named `layouts` which called a file named `main.handlebars`. I would then add my template files inside `views`. By adding `{{body}}` into my `main.handlebars`, I can inject the data from my template files into the area where I placed `{{body}}`.



The screenshot shows the Visual Studio Code interface with a workspace named 'server.js - Untitled (Workspace)'. The Explorer sidebar on the left shows a file tree for 'Assignment 1' with folders like 'models', 'node\_modules', 'public', 'css', 'img', 'video', 'views', and 'layouts'. The 'views' folder is expanded, showing 'main.handlebars', 'home.handlebars', 'listings.handlebars', 'login.handlebars', and 'registration.handlebars'. The 'main.handlebars' file is selected. The main editor displays the 'server.js' file with the following code:

```

1  const expbs = require('express-handlebars');
2  const product = require('./models/product');
3
4  app.engine('handlebars', expbs());
5  app.set('view engine', 'handlebars');
6
7  app.use(express.static('public'))
8
9
10
11
12
13
14
15
16
17
18 app.listen(3000, ()=>{
19   console.log("Web Server is up and running!") //will let us know if it even works
20 });
21
22 app.get("/",(req,res)=>{
23   res.render("home", {
24     title : "Home",
25     data : product.getAllProducts()
26   })
27 })
28
29
30 app.get("/listings",(req,res)=>{
31   res.render("listings", {
32     title : "Listings",
33     data : product.getAllMealPackages()
34   })
35 })
36

```

**Figure 7:** Displaying Express-Handlebars code that was explained previously. Also displaying views/layouts in workspace to demonstrate file tree needed for handlebars.

The screenshot shows the Visual Studio Code interface with the 'main.handlebars' file selected in the Explorer. The main editor displays the HTML content of the file, which includes a header, a hero section with a video, and a footer. The code is as follows:

```

24 </nav>
25 </div>
26 </div>
27 </header>
28
29 <div class="hero">
30   <h1 class="hero-header">Professionally Cooked Meals Right At Your Doorstep! </h1>
31   <div class="color-overlay">
32     <div class="vid-container">
33       <video autoplay loop muted>
34         <source src="/video/food_vid.mpd" type="video/mp4">
35       </video>
36     </div>
37   </div>
38 </div>
39
40 {{{body}}}
41
42 <div class="footer">
43   <div class="footer-content">
44     <div class="footer-section about">
45       <a id="bottom_of_page"></a>
46       <h1 class="logo-text"><span>LiveFitFoods</span></h1>
47       <p>
48         LiveFitFood helps makes food deliveries affordable and easy! Our goal is to brings
49         a whole new way to eat. Our in-house chefs help create nourishing dishes that get delivered right to your door!
50       </p>
51       <div class="contact">
52         <span><i class="fa fa-phone">&nbsp;&nbsp;&nbsp; 855-905-4833</i></span>
53         <span><a href="mailto:hello@livefitfoods.ca" class="contact-email"><i class="fa fa-envelope"></i>&nbsp;&nbsp;&nbsp;hello@livefitfoods.ca</a></span>
54       </div>
55       <div class="socials">
56         <a href="https://www.facebook.com/livefitfood" target="_blank"><i class="fa fa-facebook"></i></a>

```

The terminal at the bottom shows the output: 'Web Server is up and running!'.

**Figure 8:** {{{body}}} example in main.handlebars.

Finally, in our `res.render`, In the first parameter I added the name of the template file. This allows handlebars to know which template file will be used to inject data to the main layout file. In the second parameter I added variables named "title" and "data". This is because when typing those names, handlebars will find areas in our template files which contain `{{{title}}}` and `{{{data}}}`

and will implement the data I send. In other words, {{{data}}} in listings in Figure 7 will receive product.getAllProducts(). We can see where the data will be injected in Figure 8.

```

Assignment 1 > views > listings.handlebars > div.Listings_Main_Content > div.Listings-grid-container
1
2 <div class="Listings_Main_Content">
3   <div class="Listings-grid-container">
4     {{#each data}}
5       <div>
6         <div class="mptop1">
7           
8         </div>
9         <h1 class="mptitle1">{{this.Name}}</h1>
10        <p class="mpdesc1"> {{this.Desc}}
11        </p>
12      </div>
13    {{/each}}
14  </div>
15 </div>
16

```

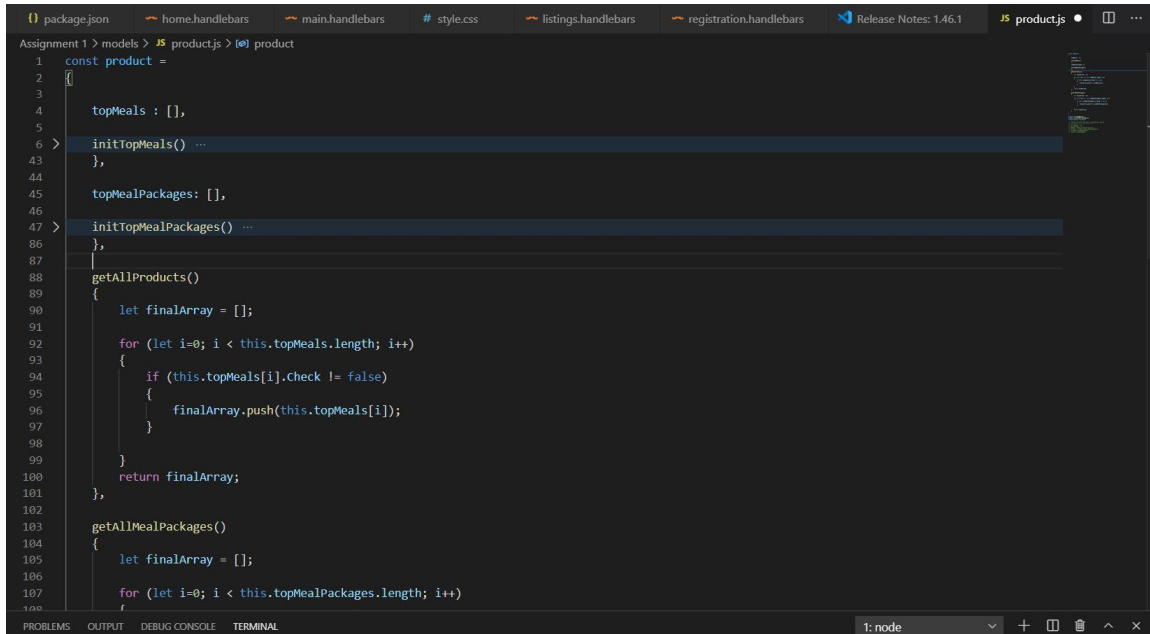
**Figure 9:** Example of where data will be injected.

Also in figure 9 we use a code from handlebars named {{#each data}}, with data being the object being passed through from server.js into listings.handlebars. #each will loop whatever content is inside the statement based on the number of elements in the data array.

## Modules

Finally, to end off the Javascript Portion of the report, I created a fake database inside a JavaScript file known as product.js. Two arrays were created known as topMeals and topMealPackages. TopMeals would be used in our home page to show the top meals that users would rate. TopMealPackages would be used in listings to show the top meal packages that users would recommend. Both arrays were placed inside a const known as const products. Products also contained four functions, with InitTopMeals and initMealPackages being used to implement data inside our two arrays, and getAllProducts and getAllMealPackages being used to only return data from the arrays if they contained a true value in the Boolean variable known as Check. I then exported products out using module.exports = product. After exporting, I would

then have to import the data using the following code in my server.js: `const product = require("./models/product")`. Product can then access the functions which can be used inside our `app.get` functions.



```

1  const product =
2  {
3
4    topMeals : [],
5
6    > initTopMeals() ...
43  },
44
45    topMealPackages: [],
46
47    > initTopMealPackages() ...
86  },
87  |
88  getAllProducts()
89  {
90    let finalArray = [];
91
92    for (let i=0; i < this.topMeals.length; i++)
93    {
94      if (this.topMeals[i].Check != false)
95      {
96        finalArray.push(this.topMeals[i]);
97      }
98    }
99
100   return finalArray;
101 },
102
103 getAllMealPackages()
104 {
105   let finalArray = [];
106
107   for (let i=0; i < this.topMealPackages.length; i++)
108   {

```

**Figure 10:** Variables inside `const product`. Note: `initTopMeals` was minimized to show the functions used to validate each element

```

product.initTopMeals();
product.initTopMealPackages();
module.exports = product;
...

```

**Figure 11:** `Module.exports = product` was used to export product out to server.js



## **HTML**

In this report, I will not dive too much into tags and what each tag does, as primarily that is Web 222 knowledge. Primarily, I will be describing unique features for each page, as well as the html used for the main layout. In total, there are four html files (template files) and one file acting as a main template, as described in the express-handlebar section in the Java Script chapter.

### **Main-Layout:**

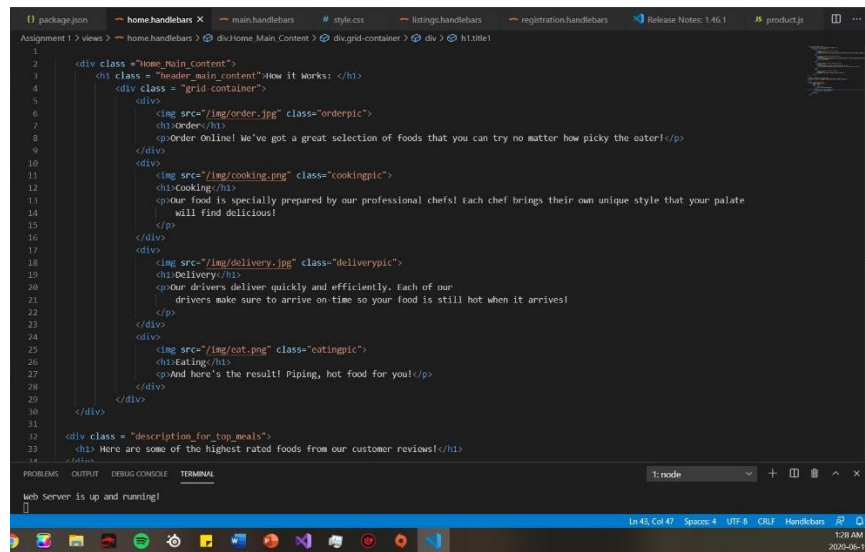
In the HTML portion, I created a navigation bar that linked towards the four pages: Home, Listings, Login, and Registration. A hero portion was created which utilized a video loop in order to display attraction for users. Main-layout also included an area for where main data should go, shown in {{{body}}} in Figure 8. A footer was also added, which included hyperlinks to various livefitfood websites (in this case, their facebook, Instagram, and twitter by utilizing awesome-font icons in css), as well as including a link towards their email. By using the syntax, mailto:, in href, I was able to allow users to automatically open their mail to write a message.

### **Home/Listings:**

In home.handlebars and listings.handlebars, I utilized the database that was imported from products.js. Code to explain specific areas, such as {{#each data}} is described in the express-handlebars section in the Java Script chapter. The difference with Home and Listings is that Home also contains information before the implementation of the database which describes how to order. This information did not utilize any database.

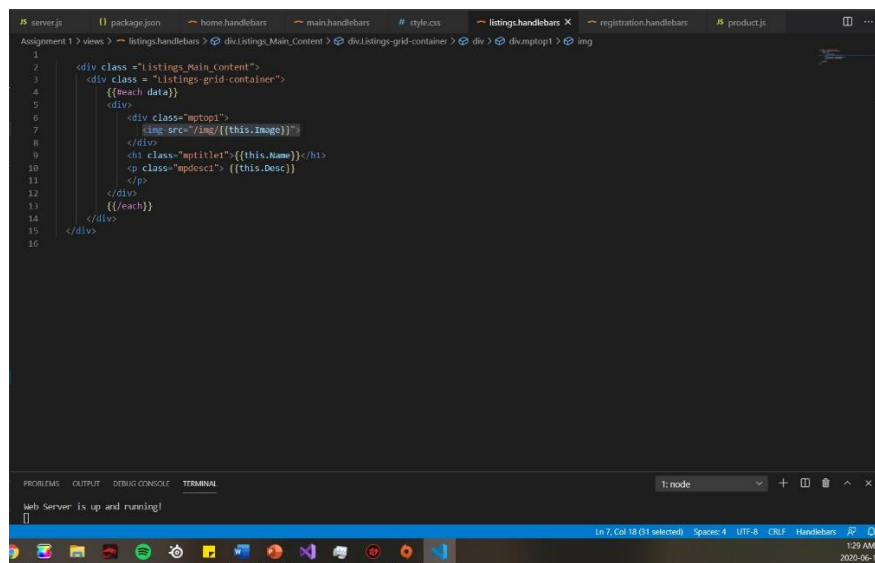
### **Login/Registration:**

Notably, both used HTML form in order to create a form for login and registration. I used a placeholder so that the user can see what they need to type in for each input. I also included Facebook, Twitter, and Instagram Icons, although they do not link to anywhere at the moment. Both contain a button tag in order to submit their information. More implementation of these forms will be shown in Assignment 2.



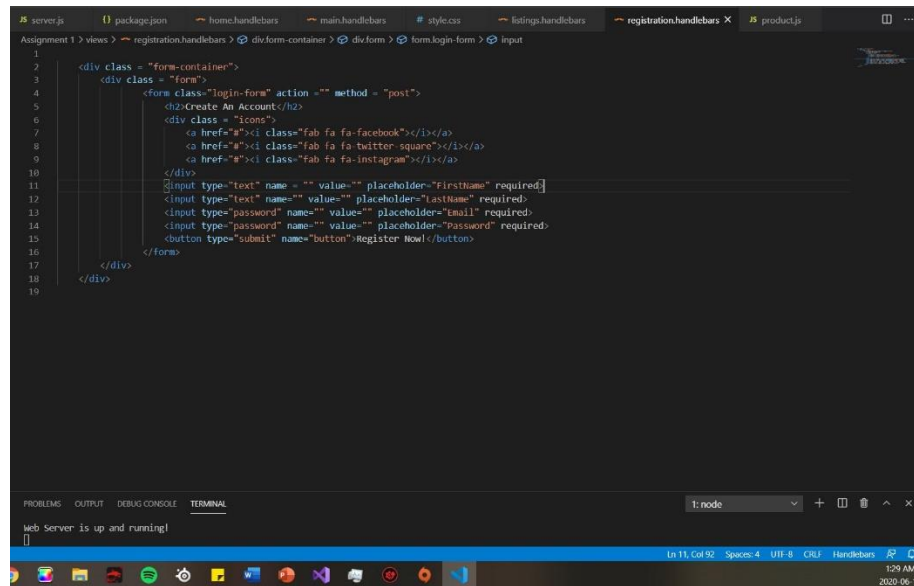
```
1 <div class="home_main_content">
2   <h1 class="header_main_content">How it Works: </h1>
3   <div class="grid-container">
4     <div>
5       
6       <h1>Order</h1>
7       <p>Order Online! We've got a great selection of foods that you can try no matter how picky the eater!</p>
8     </div>
9     <div>
10      
11      <h1>Cooking</h1>
12      <p>Our food is specially prepared by our professional chefs! Each chef brings their own unique style that your palate
13      will find delicious!
14    </p>
15    </div>
16    <div>
17      
18      <h1>Delivery</h1>
19      <p>Our drivers deliver quickly and efficiently. Each of our
20      drivers make sure to arrive on-time so your food is still hot when it arrives!
21    </p>
22    </div>
23    <div>
24      
25      <h1>Eating</h1>
26      <p>And here's the result! Piping, hot food for you!</p>
27    </div>
28  </div>
29  <div class="description_for_top_meals">
30    <h1>Here are some of the highest rated foods from our customer reviews!</h1>
31  </div>
32</div>
```

Figure 13: Example of information in home.handlebars that did not utilize a database



```
1 <div class="listings_main_content">
2   <div class="listings-grid-container">
3     {{#each data}}
4       <div>
5         <div class="mptop1">
6           
7         </div>
8         <h1 class="mptitle1">{{this.Name}}</h1>
9         <p class="mpdesc1">{{this.Desc}}</p>
10      </div>
11    </div>
12    {{/each}}
13  </div>
14</div>
```

Figure 14: Example of implementation of fake database into html



```

1  server.js  package.json  home.handlebars  main.handlebars  style.css  listings.handlebars  registration.handlebars  product.js
2  Assignment 1 > views > registration.handlebars > div.form-container > div.form > form.login-form > input
3  1
4  2  <div class = "form-container">
5  3    <div class = "form">
6  4      <form class="login-form" action="" method = "post">
7  5          <button type="button" value="Create An Account" class="button">Create An Account</button>
8  6          <div class = "icons">
9  7              <a href="#"><i class="fab fa fa-facebook"></i></a>
10 8              <a href="#"><i class="fab fa fa-twitter-square"></i></a>
11 9              <a href="#"><i class="fab fa fa-instagram"></i></a>
1210          </div>
1311          <input type="text" name="" value="" placeholder="first name" required>
1412          <input type="text" name="" value="" placeholder="last name" required>
1513          <input type="password" name="" value="" placeholder="email" required>
1614          <input type="password" name="" value="" placeholder="password" required>
1715          <button type="submit" name="button" value="Register Now!">Register Now!</button>
1816      </form>
1917    </div>
2018  </div>
2119

```

**Figure 15:** Example of the forms created. Note: login is the exact same except it only includes two inputs for email and password

## CSS

Finally, CSS was implemented in order to give an attractive look for websites. Similarly, with html, I will not go into depth with too much on CSS, as most information is based on Web 222 and is not a primary focus for Web 322. I will only go into unique CSS commands that I used that weren't commonly/never used in Web 222. NOTE: I used a CSS property known as transitions. Figures will not show transitions clearly; therefore, recommend checking transitions out while viewing the CSS section of this report.

### **Main Template:**

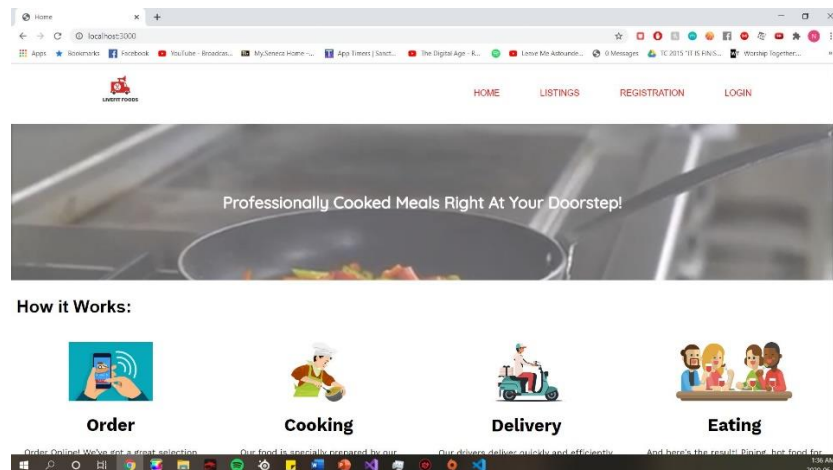
I did not use any background image, as I only used white to display a minimalistic look. I used a color template of Red, White, Black in order to have a streamline look to attractive users. I utilized a fixed position for the navigation bar, so that the navigation bar can still be used throughout the website. I also utilized a CSS property named transition, which would change the speed of what I wanted to be changed to. For instance, I added a red line above my navigation bar menus which would only show after 250ms once the cursor hovered over. I used transition throughout my web pages, specifically focusing on transition property on login and registration web pages.

**Listings/home:**

I utilized a grid format for CSS, in which I would create a grid-template-column in my grid-container. By utilizing grid-template-columns: repeat(4, 1fr), each row will display 4 columns, which are shown in the figures above for Top Meals, as well as How to Order. No transition property was used for these two webpages except for the navigation bar.

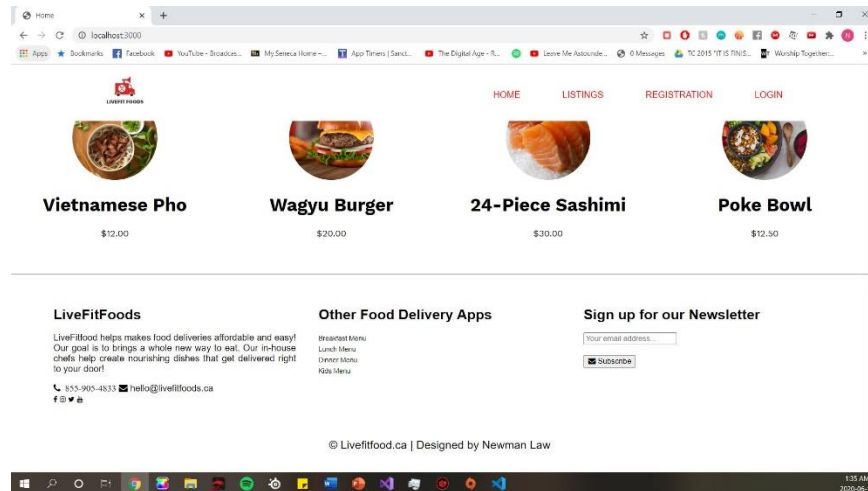
**Login/registration:**

Surprisingly, login/html had the most CSS implemented out of all the four files. I implemented images from awesome-fonts, as well as transitions for the submit button, icons for Facebook, Twitter, and Instagram, and input boxes. The transitions for the input boxes would allow the input boxes width to increase. The transitions for the submit button would make the button change color from a white background and red font, to a red background and white font. Finally, the transition for the social media icons would be changing the icons to black as well as enlarging the icons.

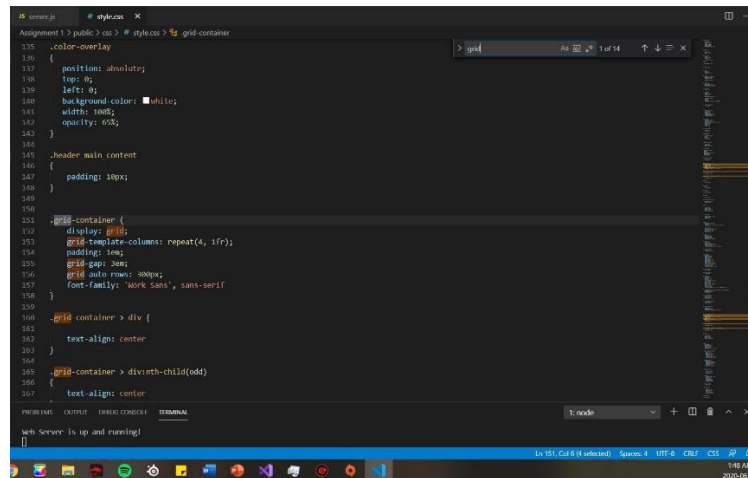


**Figure 16:** Top half of Home Page, utilizing color scheme of red, black, and white. Notice the grid layout of the four items below the Hero section. Hero section is a video; however, image is jpg.





**Figure 17:** Displaying footer, which is separated by a thin black border line. Notice the grid layout for the four items above the footer.



**Figure 18:** Utilization of CSS grid in style.css

