WEB322 Assignment 4

Submission Deadline:

Friday, March 6th, 2020 @ 11:59 PM

Assessment Weight:

9% of your final course Grade

Objective:

Build upon the code created in Assignment 3 by incorporating the Handlebars view engine to render our JSON data visually in the browser using .hbs views and layouts. Additionally, update our data-service module to allow for people to be updated using a web form.

NOTE: If you are unable to start this assignment because Assignment 3 was incomplete - email your professor for a clean version of the Assignment 3 files to start from (effectively removing any custom CSS or text added to your solution).

Specification:

As mentioned above, this assignment will build upon your code from Assignment 3. To begin, make a copy of your assignment 3 folder and open it in Visual Studio Code. Note: this will copy your .git folder as well (including the "heroku" remote for assignment 3). If you wish to start fresh with a new git repository, you will need to delete the copied .git folder and execute "git init" again in.

Part 1: Getting Express Handlebars & Updating your views

Step 1: Install & configure express-handlebars

- Use npm to install the "express-handlebars" module
- Wire up your server.js file to use the new "express-handlebars" module, ie:
    - "require" it as exphbs
    - add the app.engine() code using exphbs({ … }) and the "extname" property as ".hbs" and the "defaultLayout" property as "main" (See the Week 6 Notes)
    - call app.set() to specify the 'view engine' (See the Week 6 Notes)
- Inside the "views" folder, create a "layouts" folder

Step 2: Create the "default layout" & refactor home.html to use .hbs

- In the "layouts" directory, create a "main.hbs" file (this is our "default layout")

- Copy all the content of the "home.html" file and paste it into "main.hbs"
    - **Quick Note:** if your site.css link looks like this href="css/site.css", it must be *modified* to use a leading "/", ie href="/css/site.css"
- Next, in your main.hbs file, remove all content **INSIDE** (not including) the single … element and replace it with {{{body}}}
- Once this is done, rename home.html to home.hbs
- Inside home.hbs, remove all content **EXCEPT** what is INSIDE the single … element (this should leave a single … element containing two "columns", ie elements with class "col-md- …" and their contents)
- In your server.js file, change the GET route for "/" to "render" the "home" view, instead of sending home.html
- Test your server - you shouldn't see any changes. This means that your default layout ("main.hbs"), "home.hbs" and server.js files are working correctly with the express-handlebars module.

Step 3: Update the remaining "about", "addPeople" and "addPicture" files to use .hbs

- Follow the same procedure that was used for "home.html", for each of the above 3 files, ie:
    - Rename the .html file to .hbs
    - Delete all content **EXCEPT** what is INSIDE the single … element
    - Modify the corresponding GET route (ie: "/about", "/pictures/add" or "/people/add") to "**res.render**" the appropriate .hbs file, *instead* of using res.sendFile
- Test your server - you shouldn't see any changes, ***except*** for the fact that your menu items are no longer highlighted when we change routes (only "Home" remains highlighted, since it is the only menu item within our main.hbs "default layout" with the class "active"

Step 4: Fixing the Navigation Bar to Show the correct "active" item

- To fix the issue we created by placing our navigation bar in our "default" layout, we need to make some small updates, including adding the following middleware function ***above*** your routes in server.js:

```
app.use(function(req,res,next){
  let route = req.baseUrl + req.path;
  app.locals.activeRoute = (route == "/") ? "/" : route.replace(/\/$/, "");
  next();
});
```

  This will add the property "activeRoute" to "app.locals" whenever the route changes, ie: if our route is "/people/add", the app.locals.activeRoute value will be "/people/add".

- Next, we must use the following handlebars custom "helper" (See the Week 6 notes for adding custom "helpers")"

```
navLink: function(url, options){
  return '<li' +
    ((url == app.locals.activeRoute) ? ' class="active" ' : '') +
```

```
    '><a href="' + url + '">' + options.fn(this) + '</a></li>';
}
```

- This basically allows us to replace all of our existing navbar links, ie: About with code that looks like this {{#navLink "/about"}}About{{/navLink}}. The benefit here is that the helper will automatically render the correct <li> element add the class "active" if app.locals.activeRoute matches the provided url, ie "/about"
- Now that our helpers are in place, update ***all the navbar links*** in main.hbs to use the new helper, for example:
    - About will become {{#navLink "/about"}}About{{/navLink}}
- Test the server again - you should see that the correct menu items are highlighted as you navigate between views

Part 2: Rendering the Pictures in the "/pictures" route

Next, we'll work with pictures. It'll be easier if 1 or more pictures have been added via the application, so do this now.

Step 1: Add / configure "pictures.hbs" view and server.js

- First, add a file "pictures.hbs in the "views" directory
- Inside your newly created pictures.hbs file, add the following code to render 1 (one) of your (already-existing) uploaded pictures, ie (picture "1518186273491.jpg" - your picture will have a different timestamp):

```
<div class="row">
<div class="col-md-12">
      <h2>Pictures</h2>
      <hr />
  </div>

   <div class="col-md-4">
      <img src="/pictures/uploaded/1518186273491.jpg" class="img-responsive img-thumbnail" />
   </div>
</div>
```

  Note the classes "img-responsive" and "img-thumbnail". These are simply bootstrap classes that correctly scale and decorate the picture with a border. See https://getbootstrap.com/docs/3.3/css/#images / https://getbootstrap.com/docs/3.3/css/#images-shapes for more information

- Next, modify your GET route for /pictures. Instead of executing res.json and sending the "pictures" array, you should use res.render("pictures", **Object Containing "pictures" Array Here**); so that you can send the object containing the array of pictures as data for your "pictures" view
- Once this is complete, modify your pictures.hbs file using the handlebars #each helper to iterate over the "pictures" array, such that ***every* picture** is shown **in its own ... element** (effectively replacing our single "static" picture). This will have the effect of giving us a nice, responsive grid of multiple "col-md-4" columns, each containing its own picture.
    - **NOTE:** you can use **{{this}}** within the loop to get the current value of the item in the array of strings (this will be the filename of the current picture, ie: "1518186273491.jpg")

- If there are no pictures (ie the "pictures" array is empty), show the following element instead:

```
<div class="col-md-12 text-center">
        <strong>No Pictures Available</strong>
</div>
```

- **NOTE**: since we are hosting our app on heroku, you will notice that once the app "sleeps" and starts up again, any uploaded pictures are gone. This is expected behavior. However, if you wish to develop an application that will persist its images between restarts of the app, you can look at something like Cloudinary (this service provides a mechanism to upload pictures to Cloudinary from your server.js code and store them online using their service)

Part 3: Updating the People Route & Adding a View

Rather than simply outputting a list of people using res.json, it would be much better to actually render the data in a table that allows us to access individual people and filter the list using our existing req.params code.

Step 1: Creating a simple "People" list & updating server.js

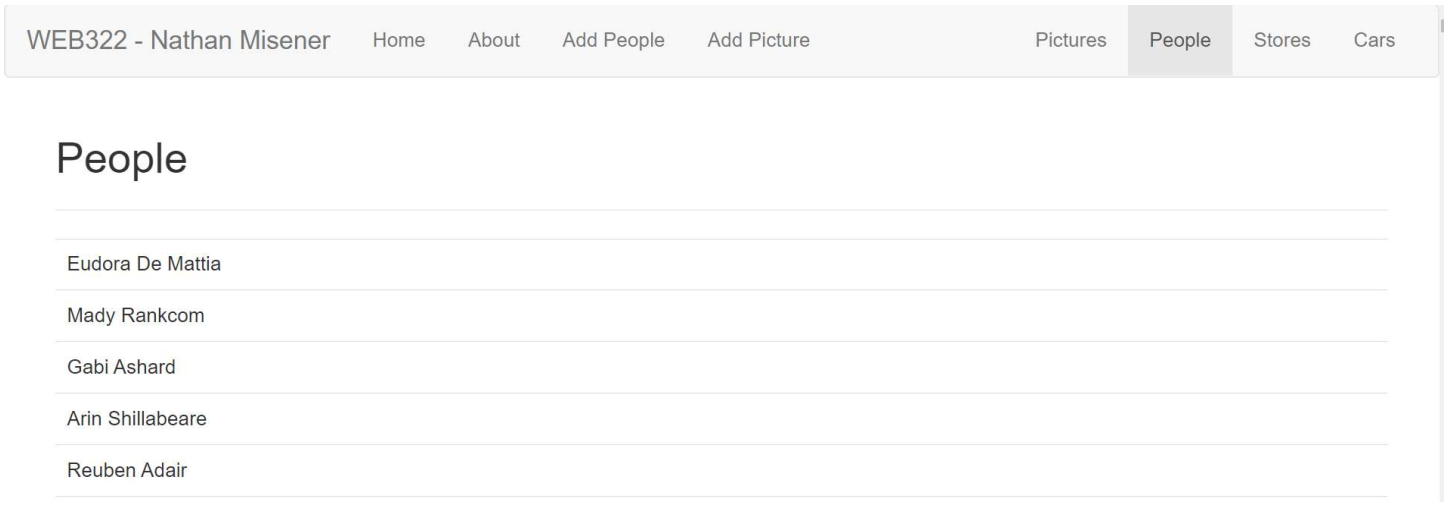
- First, add a file "people.hbs" " in the "views" directory
- Inside the newly created "people.hbs" view, add the html:

```
<div class="row">
  <div class="col-md-12">
      <h2>People</h2>
      <hr />

      <p>TODO: render a list of all People first and last names here</p>

  </div>
</div>
```

- Replace the <p> element (containing the TODO message) with code to iterate over **each person** and simply render their first and last names (you may assume that there will be an "people" array (see below).
- Once this is done, update your GET "/people" route according to the following specification
    - Every time you would have used res.json(data), modify it to instead use res.render("people", {people: data})
    - Every time you would have used res.json({message: "no results"}) - ie: when the promise has an error (ie in .catch()), modify instead to use res.render({message: "no results"});
- Test the Server - you should see the following page for the "/people" route:



Step 2: Building the Table & Displaying the error "message"

- Update the people.hbs file to render all of the data in a table, using the bootstrap classes: "table-responsive" (for the containing the table) and "table" (for the table itself) - Refer to the completed sample here https://gentle-earth-90224.herokuapp.com/people
    - The table must consist of 5 columns with the headings: **Person ID**, **Full Name**, **Phone Number**, **Address**, **and Car Vin**
    - Additionally, the Name in the Full Name column must link to /person/**id** where **id** is the person's id for that row
    - The "Phone" column must be a "tel" link to the user's phone number for that row
    - The "Car Vin" link must link to /cars?vin=**vin** where **vin** is the vin number for the person for that row
- Beneath … element, add the following code that will conditionally display the "message" only if there are no people (**HINT**: #unless people)

```
<div class="col-md-12 text-center">
        <strong>{{message}}</strong>
</div>
```

This will allow us to correctly show the error message from the .catch() in our route

Step 3: Adding new query route for /stores

Update server.js to add a new query route for stores

- /stores?retailer=***value***
    - return a JSON string consisting of all stores where ***value*** could is retailer attribute of the stores object - this can be accomplished by calling the **getStoresByRetailer(value)** function of your data-service (defined below)
- Add getStoresByRetailer(data) Function
    - In data-service.js this function will provide an array of "stores" objects whose **retailer** property matches the ***data*** parameter using the **resolve** method of the returned promise.

- If for some reason, the length of the array is 0 (no results returned), this function must invoke the **reject** method and pass a meaningful message, ie: "no results returned".

Part 4: Updating the Stores Route & Adding a View

Now that we have the "People" data rendering correctly in the browser, we can use the same pattern to render the "Stores" data in a table:

Step 1: Creating a simple "Stores" list & updating server.js

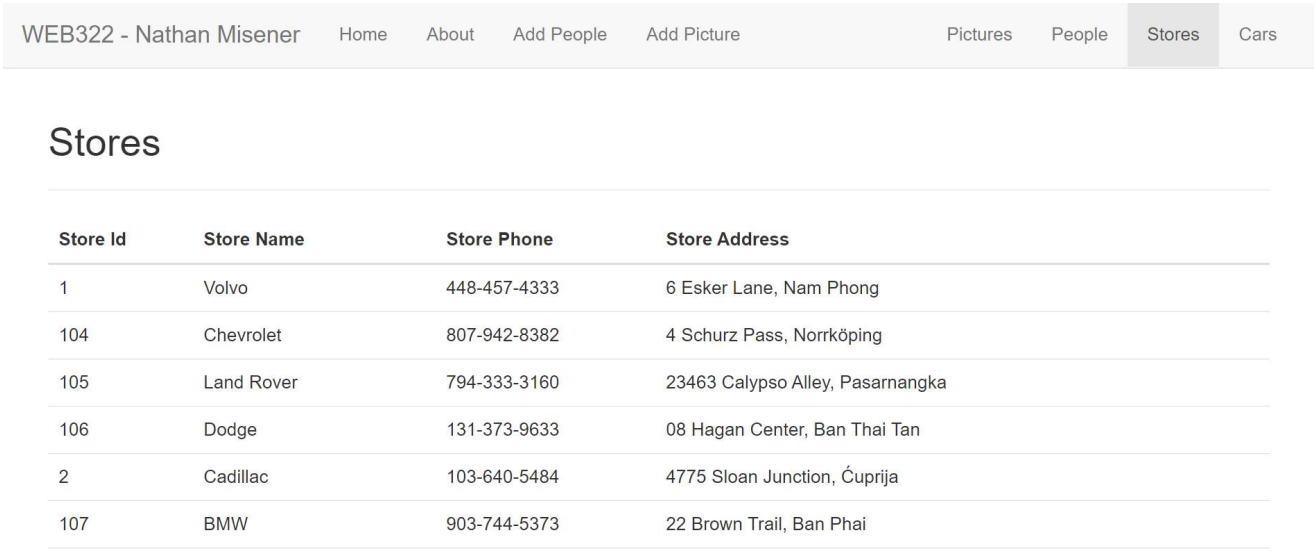
- First, add a file "stores.hbs" in the "views" directory
- Inside the newly created " stores.hbs" view, add the html:

```
<div class="row">
  <div class="col-md-12">
      <h2>Stores</h2>
      <hr />

      <p>TODO: render a list of all stores id's and names here</p>

  </div>
</div>
```

- Replace the <p> element (containing the TODO message) with code to iterate over **each store** and simply render their **id**, **name, phone and address** values (you may assume that there will be a "stores" array (see below).
- Once this is done, update your GET "/stores" route according to the following specification
    - Instead of using res.json(data), modify it to instead use res.render("stores", {stores: data});
- Test the Server - you should see the following page for the "/stores" route:



Step 2: Building the Table

- Update the stores.hbs file to render all of the data in a table, using the bootstrap classes: "table-responsive" (for the containing the table) and "table" (for the table itself).
- The table must consist of 4 columns with the headings: **Store id, Store Name, Store Phone, and Address**
- The "Store Name" link must link to /cars?make=**make** where **make** is the make of the retailer for that row
- The "Phone" column must be a "tel" link to the user's phone number for that row
- Refer to the example online at https://gentle-earth-90224.herokuapp.com/stores

Part 5: Updating the Cars Route & Adding a View

Now that we have the "People" and “Stores” data rendering correctly in the browser, we can use the same pattern to render the "Cars" data in a table:

Step 1: Creating a simple "Cars" list & updating server.js

- First, add a file "cars.hbs" in the "views" directory
- Inside the newly created " cars.hbs" view, add the html:

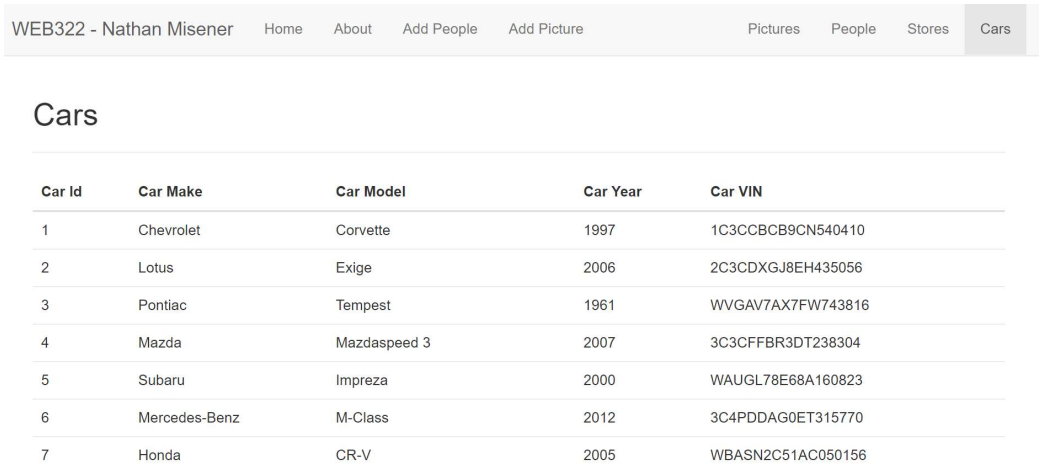
```
<div class="row">
  <div class="col-md-12">
      <h2>Cars</h2>
      <hr />

      <p>TODO: render a list of all cars id's and names here</p>

  </div>
</div>
```

- Replace the <p> element (containing the TODO message) with code to iterate over **each car** and simply render their **id**, **make, model, year and vin** values (you may assume that there will be a "cars" array (see below).
- Once this is done, update your GET "/cars" route according to the following specification
    - Instead of using res.json(data), modify it to instead use res.render("cars", {cars: data});

- Test the Server - you should see the following page for the "/cars" route:



Step 2: Building the Table

- Update the cars.hbs file to render all of the data in a table, using the bootstrap classes: "table-responsive" (for the containing the table) and "table" (for the table itself).
- The table must consist of 5 columns with the headings: **Car id, Car Make, Car Model, Car Year and Vin**
- The "Car make" link must link to /stores?retailer=**retailer** where **retailer** is the retailer of the make for that row
- The "Car Year" " link must link to /cars?year=**year** where **year** is the year of the car for that row
- The "Car Vin" link must link to /people?vin=**vin** where **vin** is the vin of the car for that row
- Refer to the example online at https://gentle-earth-90224.herokuapp.com/cars

Part 6: Updating Existing People

The last piece of the assignment is to create a view for a single person. Currently, when you click on an person's name in the "/people" route, you will be redirected to a page that shows all of the information for that person as a JSON-formatted string (ie: accessing **http://localhost:8080/person/21**, should display a JSON formatted string representing the corresponding person - person 21).

Now that we are familiar with the express-handlebars module, we should add a view to render this data in a form and allow the user to save changes.

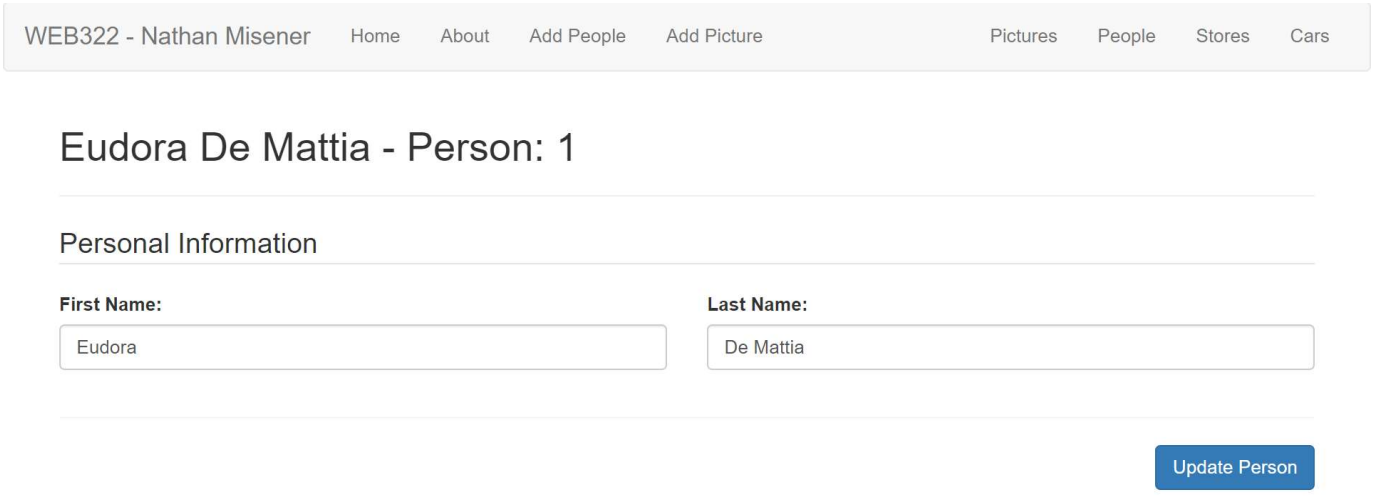
Step 1: Creating new .hbs file / route to Update Person

- First, add a file "person.hbs" in the "views" directory
- Inside the newly created "person.hbs" view, add the html (**NOTE:** Some of the following html code may wrap across lines to fit on the .pdf - be sure to check that the formatting is correct after pasting the code):

```
<div class="row">
  <div class="col-md-12">
    <h2>{{person.first_name}} {{ person.last_name}} - Person: {{ person.id}}</h2>
    <hr />
```

```
      <form method="post" action="/person/update">
        <fieldset>
          <legend>Personal Information</legend>
          <div class="row">
            <div class="col-md-6">
              <div class="form-group">
                <label for="first_name">First Name:</label>
                <input class="form-control" id="first_name" name="first_name" type="text"
value="{{person.first_name}}" />
              </div>
            </div>
            <div class="col-md-6">
              <div class="form-group">
                <label for="last_name">Last Name:</label>
                <input class="form-control" id="last_name" name="last_name" type="text"
value="{{person.last_name}}" />
              </div>
            </div>
          </div>
        </fieldset>
        <hr />
        <input type="submit" class="btn btn-primary pull-right" value="Update Person" /><br /><br /><br />
      </form>
    </div>
  </div>
```

- Once this is done, update your GET "/person/:id" route according to the following specification
    - Use **res.render("person", { person: data });** inside the **.then()** callback (instead of res.json) and use **res.render("person",{message:"no results"});** inside the **.catch()** callback
- Test the server **(/person/1**) - this will get you started on creating / populating the form with user data:



- Continue this pattern to develop the full form to match the completed sample here - you may use the code in the sample to help guide your solution
    - **Id**: type: "hidden", name: "id"
    - **Phone**: type: "text", name: "phone"
    - **Address (Street):** type: "text", name: "address"
    - **Address (City):** type: "text", name: "city"
    - **Person's Vin Number:** type: "text", name: "vin"
- No validation (client or server-side) is required on any of the form elements at this time
- Once the form is complete, we must add the **POST** route: **/person/update** in our server.js file:

```
app.post("/person/update", (req, res) => {
  console.log(req.body);
  res.redirect("/people");
});
```

This will show you all the data from your form in the console, once the user clicks "Update Person". However, in order to take that data and update our "people" array in memory, we must add some new functionality to the **data-service.js** module:

Step 2: Updating the data-service.js module

- Add the new method: **updatePerson(personData)** that **returns a promise**. This method will:
    - Search through the "people" array for a person with an **id** that matches the JavaScript object (parameter personData).
    - When the matching person is found, **overwrite** it with the new person passed into the function (parameter personData)
    - Once this has completed successfully, invoke the **resolve()** method without any data.
- Now that we have a new **updatePerson()** method, we can invoke this function from our newly created **app.post("/person/update", (req, res) => { … });** route. Simply invoke the **updatePerson**() method with the **req.body** as the parameter. Once the promise is resolved use the **then()** callback to execute the **res.redirect("/people");** code.
- Test your server in the browser by updating Person 21 (Regina Dunphy). Once you have clicked "Update Person" and are redirected back to the People list, Person 21 should show your changes!

Part 7: Pushing to Heroku

Once you are satisfied with your application, deploy it to Heroku:

- Ensure that you have checked in your latest code using **git** (from within Visual Studio Code)
- Open the integrated terminal in Visual Studio Code
- Log in to your Heroku account using the command **heroku login**

- Create a new app on Heroku using the command **heroku create**
- Push your code to Heroku using the command **git push heroku master**
- **IMPORTANT NOTE:** Since we are using an "**unverified" free** account on Heroku, we are limited to only **5 apps**, so if you have been experimenting on Heroku and have created 5 apps already, you must delete one (or verify your account with a credit card). Once you have received a grade for Assignment 1, it is safe to delete this app (login to the Heroku website, click on your app and then click the **Delete app…** button under "**Settings**").

Testing: Sample Solution

To see a completed version of this app running, visit: https://gentle-earth-90224.herokuapp.com/

Please note: This solution is **visible** to **ALL students** and **professors** at Seneca College. It is your responsibility as a student of the college not to post inappropriate content / images to the shared solution. It is meant purely as an exemplar and any misuse will not be tolerated.

Assignment Submission:

- Before you submit, consider updating **site.css** to provide additional style to the pages in your app. Black, White and Gray is boring, so why not add some cool colors and fonts (maybe something from Google Fonts)? This is your app for the semester, you should personalize it!
- Next, Add the following declaration at the top of your **server.js** file:

```
/*********************************************************************************
* WEB322 – Assignment 04
* I declare that this assignment is my own work in accordance with Seneca  Academic Policy.  No part
* of this assignment has been copied manually or electronically from any other source
* (including 3rd party web sites) or distributed to other students.
*
* Name: _____ Student ID: _____ Date: _____
*
* Online (Heroku) Link: _____
*
********************************************************************************/
```

- Compress (.zip) your web322-app folder and submit the .zip file to My.Seneca under **Assignments** -> **Assignment** 4

Important Note:

- **NO LATE SUBMISSIONS** for assignments. Late assignment submissions will not be accepted and will receive a **grade of zero (0)**.
- After the end (11:59PM) of the due date, the assignment submission link on My.Seneca will no longer be available.
- Submitted assignments must run locally, ie: start up errors causing the assignment/app to fail on startup will result in a **grade of zero (0)** for the assignment.