WEB322 Assignment 2

Submission Deadline:

Friday, June 3th, 2022 @ 11:59 PM

Assessment Weight:

9% of your final course Grade

Objective:

Create and publish a web app that uses multiple routes which serve static files (HTML & CSS) as well as create a "blog service" module for accessing data. This will serve as the "scaffolding" for future assignments.

Specification:

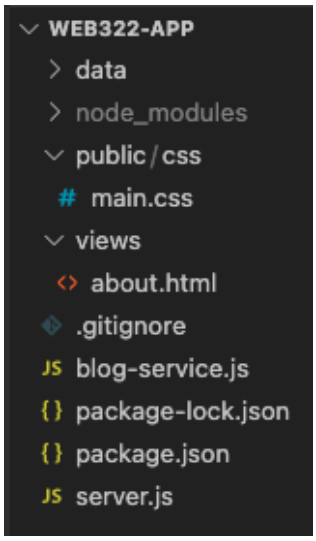
This assignment will involve creating multiple routes that serve a specific HTML page & JSON data.

Part 1: Dev Environment, Files & Folder Structure

Step 1: Development Environment

- Create a folder called **web322-app**. This will serve as our main application that we will be updating and modifying throughout this course.
- Inside this folder, initialize a local **Git repository (**using **git init** from the integrated terminal**)**
- Add the file **server.js**
- Create a **package.json** file using **npm init**. Ensure that your "entry point" is **server.js** (this should be the default), and "author" is your full name, ie: "John Smith"
- Add the file **blog-service.js**
- Add the file **.gitignore** containing the single line**: /node_modules**
  (This will prevent git from tracking changes in the node_modules folder)
- Obtain the **Express.js** module using **npm install express**
- **Commit** your changes your **local git repository** (using the source control icon showing the number of changes, ie: 5) with the message "initial commit"

Step 2: Adding Files / Folders

- 

- Add the folder **views** - this will be the location of the .html files that we will be using in our application
- Add the folder **public** - this will be the location of the .css, client side .js & image files that we use in our application
- Add the folder **data** - this will be a temporary source of static data (JSON) for our application
- Inside the **views** folder, add the file **about.html**
- Inside the **public** folder, add the folder **css**
- Inside the **public/css** folder - add the file **main.css** (this will serve as the main .css file for our app)
- Once you commit to your latest changes, your folder structure should look like the image on the right:

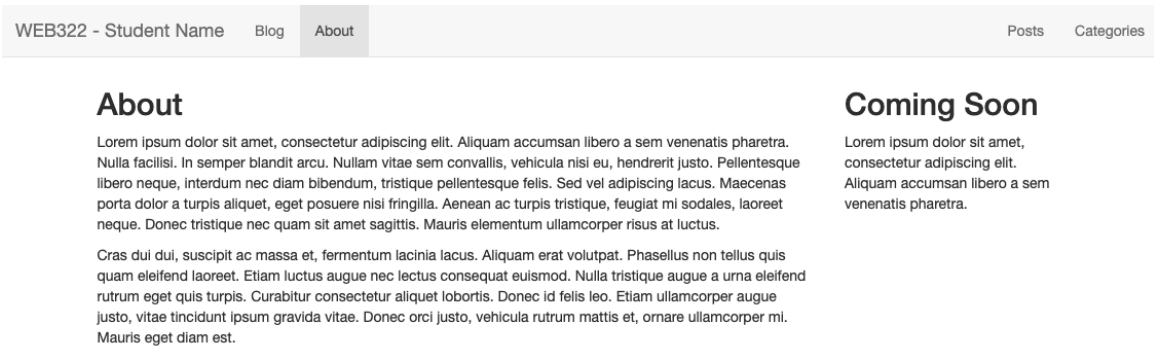
Step 3: Adding Static Content (about.html)

- Before starting on your **server.js** file, add some html **about.html** using the following template**:** https://scs.senecac.on.ca/~patrick.crawford/shared/winter-2022/web322/A2/template.html.txt - this leverages the Bootstrap 3 & jQuery libraries (discussed in detail during Week 11)
- At this point, we must make some changes to the page (what's currently there is only a starting point)
    - Update "Link 1" to read "Blog" and change the link's "href" property from "#" to "/blog"
    - Update "Link 2" to read "About" and change the link's "href" property from "#" to "/about"
    - Update the page "title" to read "About"
    - Ensure the heading (h2) for the left column reads "About"
        - **NOTE**: Please feel free to update the "Lorem Ipsum" text to something more personal, ie: what you're currently studying, interests, goals, etc.
    - Ensure the heading (h2) for the right column reads "Coming Soon"

        - **NOTE:** You may also updated the sample text for something more fun, ie: "Stay tuned for future updates…", etc.
    - Modify the "navbar-brand" span element to read "WEB322 - Student Name" where "Student Name" is your name, ie "John Smith", etc
    - Update "Link 3" to read "Posts" and change it's "href property from "#" to "/posts"
    - Update "Link 4" to read "Categories" and change it's "href property from "#" to "/categories"

Step 4: Update server.js & testing the app

- Now that all the files are in place, update your **server.js** file according to the following specifications (**HINT**: Refer to the sample code from **week 2** for reference):
    - The server must make use of the "**express**" module
    - The server must listen on **process.env.PORT || 8080**
    - The server must output: "Express http server listening on ***port***" - to the console, where ***port*** is the port the server is currently listening on (ie: 8080)
    - The route "**/**" must **redirect** the user to the "**/about**" route – this can be accomplished using **res.redirect()** (see week 4 "Response object")
    - The route "**/about**" must return the **about.html** file from the **views** folder
    - **NOTE**: for your server to correctly return the "/css/main.css" file, the "**static**" middleware must be used: in your **server.js** file, add the line: **app.use(express.static('public'));** before your "routes" (see week 4 "Serving static files")
    - From the integrated terminal, enter the command **node server.js** and verify the following:
        - The integrated terminal shows "Express http server listening on 8080"
        - The url: **http://localhost:8080** should redirect to the "About" page:



    - **NOTE**: At this point, if you wish to make any updates to the look and feel of the site, please feel free to update your **main.css** to personalize it.

      You can also **use a different Bootstrap 3 theme** by browsing to https://bootswatch.com/3

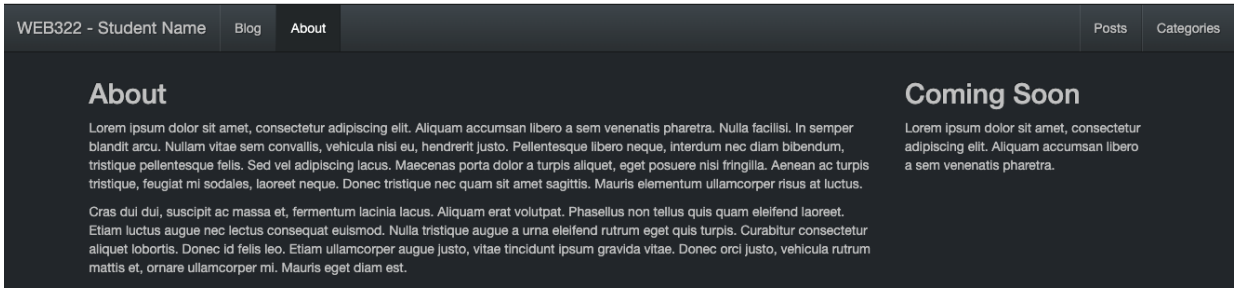
      They have 16 free templates that can be used by simply swapping the bootstrap CSS for the CSS available by clicking the "download" button for each of the themes.

      For example we can use the "Slate" theme (below) by commenting out our Bootstrap CSS and adding the CSS for the "Slate" theme, ie:

      <!-- <link rel="stylesheet"
      href="https://stackpath.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css"
      integrity="sha384-
      HSMxcRTRxnN+Bdg0JdbxYKrThecOKuH5zCYotlSAcp1+c8xmyTe9GYg1l9a69psu"

crossorigin="anonymous"> -->

<link rel="stylesheet" href="https://bootswatch.com/3/slate/bootstrap.min.css" />



Part 2: Blog Service, Blog, Posts & Categories

Step 1: Obtaining the Data

- Create 2 new files inside the "data" folder: **posts.json** and **categories.json**
- Open your web browser and navigate to: this link (categories.json) and copy the contents of the JSON file to your own categories.json file (within the "data" folder).
- Next, navigate to: this link (posts.json) and copy the entire contents of the JSON file to your own posts.json file (within the "data" folder) - this should be an array of 30 "post" objects

Step 2: Updating the custom blog-service.js module

- The file that we added at the beginning of this assignment ("blog-service.js") is going to be a module that we will use within our server.js file.
- Your first step is to "**require**" this module at the top of your **server.js** file so that we can use it to interact with the data from server.js

Step 3: Adding additional Routes:

For now, we will be making use of this blog data by sending it back to the client using specific routes. One route (ie: the "/blog" route) will serve as the public-facing part of our application (along with "/about"), whereas other routes will be used in the future for dealing with managing the blog (ie: creating / editing posts, adding categories, etc) in a private area (later protected by a login page & user authentication).

Inside your server.js add routes to respond to the following "get" requests for the application. Once you have written the routes, test that they work properly by returning a confirmation string using **res.send()** and testing the server using localhost:8080.

For example, **localhost:8080/blog** could be set up to return something like **"TODO: get all posts who have published==true"**. This will help to confirm that your routes are set up properly *before* they return real data.

/blog

- This route will return a JSON formatted string containing all of the posts within the posts.json file whose **published** property is set to **true** (ie: "published" posts).

/posts

- This route will return a JSON formatted string containing all the posts within the posts.json files

/categories

- This route will return a JSON formatted string containing all of the categories within the categories.json file

[no matching route]

- If the user enters a route that is not matched with anything in your app (ie: http://localhost:8080/app) then you must return the custom message "**Page Not Found**" with an HTTP status code of **404**.
- **Note:** at this point, you may wish to send a custom 404 page back to the user (completely optional, but everyone loves a good 404 page. Here's some ***inspiration*** for your own designs: https://www.creativebloq.com/web-design/best-404-pages-812505

Step 4: Writing the blog-service.js module:

The promise driven blog-service.js module will be responsible for reading the posts.json and categories.json files from within the "data" directory on the server, parsing the data into arrays of objects and returning elements (ie: "posts" objects) from those arrays to match queries on the data.

Essentially the blog-service.js module will encapsulate all the logic to work with the data and only expose accessor methods to fetch data/subsets of the data.

Module Data

The following two arrays should be declared "globally" within your module:

- **posts** - type: **array**
- **categories** - type: **array**

Exported Functions

Each of the below functions are designed to work with the posts and categories datasets. Unfortunately, we have no way of knowing how long each function will take (we cannot assume that they will be instantaneous, ie: what if we move from .json files to a remote database, or introduce hundreds of thousands of objects into our .json dataset? - this would increase lag time).

Because of this, **every one of the below functions must return a promise** that **passes the data** via it's "**resolve**" method (or - if **no data was returned**, passes an **error message** via it's "**reject**" method).

When we access these methods from the server.js file, we will be assuming that they return a promise and we will respond appropriately with **.then()** and .**catch()** (see "Updating the new routes…" below).

initialize()

- This function will read the contents of the "./data/posts.json" file

  hint: see the fs module & the fs.readFile method, ie (from the documentation):

  ```
  const fs = require("fs"); // required at the top of your module

  fs.readFile('somefile.json', 'utf8', (err, data) => {
      if (err) throw err;
      console.log(data);
  });
  ```

  Do not forget convert the file's contents into an array of objects instead of plain text (**hint**: see JSON.parse) , and assign that array to the **posts array** (from above).

- Only once the read operation for "./data/posts.json" has completed successfully (not before), repeat the process for the "./data/categories.json" and assign the parsed object array to the **categories array** from above.
- Once these two operations have finished successfully, invoke the **resolve** method for the promise to communicate back to server.js that the operation was a success.
- If there was an error at any time during this process, **instead** of throwing an error, invoke the **reject** method for the promise and pass an appropriate message, ie: reject("unable to read file").

getAllPosts()

- This function will provide the full array of "posts" objects 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".

getPublishedPosts()

- This function will provide an array of "post" objects whose **published** property is **true** 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".

getCategories()

- This function will provide the full array of "category" objects 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".

Step 5: Updating the code surrounding app.listen()

Before we start updating the routes in server.js to use our new blog-service module, we must make a small update to the code ***surrounding*** the app.listen() call at the bottom of the server.js file. This is where the **initialize()** method from our blog-service.js module comes into play.

Fundamentally, initialize() is responsible for reading the .json files from the "data" folder and parsing the results to create the "global" (to the module) arrays, "posts" and "categories" that are used by the other functions. However, it also returns a **promise** that will only **resolve** successfully once the files were read correctly and the "posts" and "categories" arrays were correctly loaded with the data.

Similarly, the promise will **reject** if any error occurred during the process. Therefore, we must **only call app.listen()** if our call to the **initialize()** method is successful, ie: .**then(() => { //start the server })**.
If the initialize() method invoked **reject**, then we should not start the server (since there will be no data to fetch) and instead a meaningful error message should be sent to the console, ie: **.catch(()=>{ /*output the error to the console */})**

Step 6: Updating the new routes to use blog-service.js

Now that the blog-service.js module is complete, we must update our new routes (ie: /blog, /posts & /categories) to make calls to the service and fetch data to be returned to the client.

Since our blog-service.js file exposes functions that are guaranteed to return a **promise** that (if resolved successfully), will contain the requested data, we must make use of the **.then()** method when accessing the data from within our routes.

For example, the **/categories** route must make a call to the **getCategories()** method of the blog-service.js module to fetch the correct data. If **getCategories()** was successful, we can use **.then((data) => { /* send data to the client */ })** to access the data from the function and send the response back to the client.

If any of the methods were unsuccessful, however, the. then**()** method will not be called - the **catch()** method will be called instead. If this is the case, the server **must** return a simple object with 1 property: "message" containing the message supplied in the **.catch()** method, ie: **.catch((err) => { /* return err message in the format:** {message: ***err***} ***/ })**.

By **only** calling **res.send()** (or **res.json()**) from within **.then()** or **.catch()** we can ensure that the data will be in place (no matter how long it took to retrieve) before the server sends anything back to the client.

Step 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**").

Step 8. Pushing code to a private GitHub repository

Once you have pushed your code from local to Heroku, **you must also push to a remote repository:**

- Create a new private repository in GitHub and name it **web322-app**
- Follow “Connect to GitHub” instructions: https://kbroman.org/github_tutorial/pages/init.html

- Share this repository by going to your **web322-app** repository in GitHub -> Settings -> Collaborators -> Add People -> Enter "at-seneca" and **Invite Collaborator**

Testing: Sample Solution

To see a completed version of this app running, visit: https://web322-a2-sample.herokuapp.com

Assignment Submission:

- If you haven't already, please consider updating **main.css** to or using one of the themes from https://bootswatch.com/3 to provide additional style to the pages in your app. Black, White and Gray can be boring, so why not add some cool colors and fonts (maybe something from Google Fonts)? This is your app for the semester, feel free to personalize it!
- Next, Add the following declaration at the top of your **server.js** file:

```
/*********************************************************************
**********
* WEB322 – Assignment 02
* 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** 2

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.