

BTI325 Assignment 2

Submission Deadline:

Section NAA: Tuesday, Oct 1st, 2019 @ 11:59 PM

Section NBB: Thursday, Oct 3rd, 2019 @ 11:59 PM

Section NCC: Monday, Sep 30th, 2019 @ 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 "data service" module for accessing data. This will serve as the "scaffolding" for future assignments.

Specification:

This assignment will involve creating multiple routes that serve specific HTML pages & JSON.

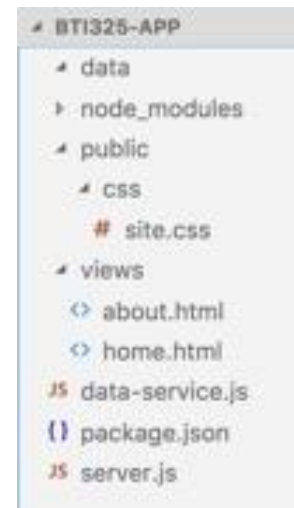
Part 1: Dev Environment, Home & About

Step 1: Development Environment

- Create a folder called **bti325-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**
- Add the file **data-service.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"
- Obtain the **Express.js** module using **npm install express --save**
- **Commit** your changes to your **local git repository** (using the source control icon showing the number of changes) 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 files **home.html** and **about.html**
- Inside the **public** folder, add the folder **css**
- Inside the **public/css** folder - add the file **site.css** (this will serve as the main .css file for our app)
- Your folder structure should now look like the image to the right:



Step 3: Adding Static Content (home.html & about.html)

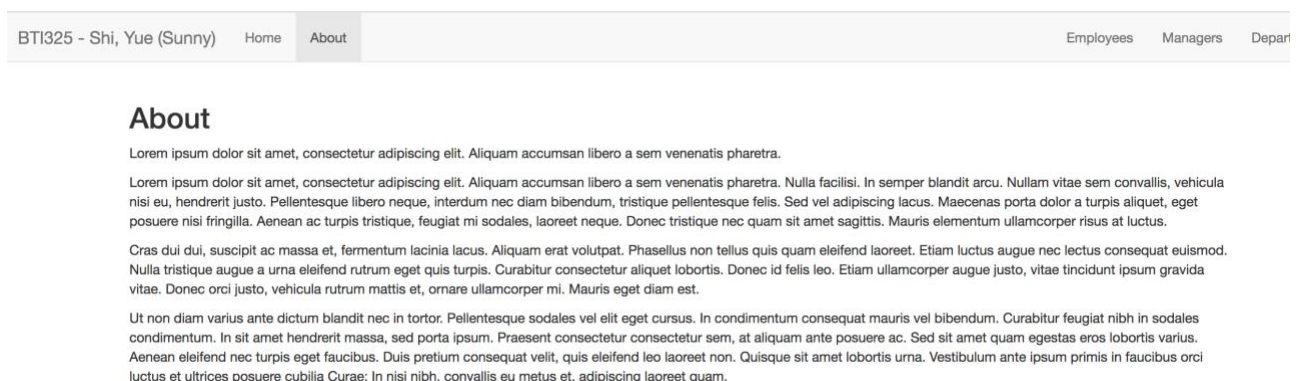
- Before starting on your **server.js** file, add some html to **home.html** & **about.html** using the following template for both files: <https://github.com/sunnyDay100/BTI325-F19/blob/master/A2/A2-template.html> - this leverages the Bootstrap 3 & jQuery libraries (discussed in detail during Week 11). Test using Firefox & Chrome. It's OK that Safari shows differently.
- At this point, both files should be exactly the same, however we must make some changes to each page (what's currently there is only a starting point)
 - **home.html**
 - Update "Link 1" to read "Home"
 - Update "Link 2" to read "About" and change its "href" property from "#" to "/about":
 - Update the page "title" to read "Home"
 - Ensure the heading (h2) for the left column reads "Coming Soon" - and provide a relevant message to the user
 - Ensure the heading (h2) for the right column reads "Welcome"
 - **about.html**
 - Update "Link 1" to read "Home", remove the class "active" from the parent element and change the link's "href" property from "#" to "/"
 - Update "Link 2" to read "About" and add the class "active" to the parent element
 - Update the page "title" & heading (h2) to read "About"
 - Modify the grid layout from 2 columns to 1 column (class="col-md-12"). We'll discuss it in lab. Also, you can reference week 11 "Responsive Grid System" – <https://web322.ca/notes/week11>
 - **both home.html & about.html**
 - modify the "navbar-brand" span element to read "BTI325 - Student Name" where "Student Name" is your name, ie "John Smith", etc
 - Update "Link 3" to read "Employees" and change its "href" property from "#" to "/employees"
 - Update "Link 4" to read "Managers" and change its "href" property from "#" to "/managers"
 - Update "Link 5" to read "Departments" and change its "href" property from "#" to "/departments"

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 return the **home.html** file from the **views** folder
 - The route **"/about"** must return the **about.html** file from the **views** folder
 - NOTE**: for your server to correctly return the "css/site.css" file, the **"static"** middleware must be used: in your **server.js** file, add the line: **app.use(express.static('public'))**; before your "routes" - we will discuss this in greater detail in Week 4
 - 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> shows the "Home" page:



- The url: <http://localhost:8080/about> shows the "About" page



Part 2: Data Service, Employees, Managers & Departments

Step 1: Obtaining the Data

- Create 2 new files inside the "data" folder: **departments.json** and **employees.json**
- Open your web browser and navigate to <https://github.com/sunnyDay100/BTI325-F19/blob/master/A2/departments.json>
Copy the contents of the JSON file to your own **departments.json** file (within the "data" folder).
- Next, navigate to <https://github.com/sunnyDay100/BTI325-F19/blob/master/A2/employees.json>

Copy the entire contents of the JSON file to your own **employees.json** file (within the "data" folder) - this should be an array of 280 "employee" objects

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

- The file that we added at the beginning of this assignment ("**data-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:

We will be making use of this employee data from a different location from our "/" and "/about" routes. These routes will serve as the public-facing pieces of our application, whereas we will be dealing with employee management in a private area (later protected by a login page / user authentication, etc).

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/managers** could be set up to return something like "**TODO: get all employees who have isManager==true**". This will help to confirm that your routes are set up properly.

Important Note: Any response sending JSON from the server must include the correct content-type header - see [res.json\(\[body\]\)](#)

[Details to implement the following routes are available below in Step 4](#)

/employees

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

/managers

- This route will return a JSON formatted string containing all the employees whose **isManager** property is set to **true**.

/departments

- This route will return a JSON formatted string containing all of the departments within the departments.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: <https://medium.com/@CollectUI/404-page-design-inspiration-march-2017-f6d9f7efd054>)

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

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

- **employees** - type: **array**
- **departments** - type: **array**

Exported Functions

Each of the below functions are designed to work with the employees and departments datasets. Since 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 its "**resolve**" method (or - if **no data was returned**, passes an **error message** via its "**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/employees.json"** file (hint: see the [fs module](#) & the [fs.readFile](#) method), convert the file's contents into an array of objects (hint: see [JSON.parse](#)), and assign that array to the **employees array** (from above).
- Only once the read operation for **"./data/employees.json"** has completed successfully (not before), repeat the process for the **"./data/departments.json"** and assign the parsed object array to the **departments 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, invoke the **reject** method for the promise and pass an appropriate message, ie: **reject("unable to read file")**.

getAllEmployees()

- This function will provide the full array of "employee" 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".

getManagers()

- This function will provide an array of "employee" objects whose **isManager** 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".

getDepartments()

- This function will provide the full array of "department" 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 data-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 data-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, "employees" and "departments" 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 "employees" and "departments" 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 data-service.js

Now that the **data-service.js** module is complete, we must update our new routes (ie: **/employees, /managers & /departments**) to make calls to the service and fetch data to be returned to the client. Recall: Any response sending JSON from the server must include the correct content-type header - see [res.json\(\[body\]\)](#).

Since our **data-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 **/departments** route must make a call to the **getDepartments()** method of the data-service.js module to fetch the correct data. If **getDepartments()** was successful, we can use **.then((data) => { /*return JSON data*/ })** 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 JSON object with 1 property: "message" containing the message supplied in the `.catch()` method, ie: `.catch(err) => { /* return err message in the JSON format: {message: err} */ }`.

By **only** calling `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**").

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:

```
/* *****  
* BTI325 – Assignment 2  
* 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 bti325-app folder and submit the .zip file to My.Seneca under **Assignments -> A2**
- Late submission will be penalized with 10% of this assignment marks for each school day up to 5 school days, after which it will receive 0 marks.