BTI325 Assignment 4

Submission Deadline: Sunday Nov 8, 2020 @ 11:59 PM

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 employees to be updated using a web form.

NOTE: If you are unable to start this assignment because Assignment 3 was incomplete - email me 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 be built 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 the variable *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 and examples)
    - call app.set() to specify the 'view engine' (See the Week 6 Notes and examples)
- 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** (excluding) 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. Do not need to pass any data to "home" view.
- 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", "addEmployee" and "addImage" 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", "/images/add" or "/employees/add") to "**res.render**" the appropriate .hbs file, *instead* of using res.sendFile. Do not need to pass any data to the "about" view.
- 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 "/employees/add", the app.locals.activeRoute value will be "/employees/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"

- Next, while we're adding custom "helpers" let's add one more that we will need later:

```
equal: function (lvalue, rvalue, options) {
    if (arguments.length < 3)
        throw new Error("Handlebars Helper equal needs 2 parameters");
    if (lvalue != rvalue) {
        return options.inverse(this);
    } else {
        return options.fn(this);
    }
}
```

This helper will give us the ability to evaluate conditions for equality, ie {{#equals "a" "a"}} … {{/equals}} will render the contents, since "a" equals "a". It's exactly like the "if" helper, but with the added benefit of evaluating a simple expression for equality

- 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}}
    - **NOTE**: You can remove the "/managers" menu item from main.hbs and the "/managers" route from server.js, as we will not be using these
- Test the server again - you should see that the correct menu items are highlighted as you navigate between views

Part 2: Rendering the Images in the "/images" route

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

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

- First, add a file "images.hbs in the "views" directory
- Inside your newly created images.hbs file, add the following code to render 1 (one) of your (already-existing) uploaded images, ie (image "1518186273491.jpg" - **your image will have a different name, the code below should change image name correspondingly in <img src=…>**):

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

    <div class="col-md-4">
        <img src="/images/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 image 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 /images. Instead of executing res.json and sending the "images" array of file names, we'll display the array of images. First try res.render("images"); This is to test your route "images". You should see your above example picture taking 1/3 of the horizontal screen space.

  Next to display the array of images, which are obtained from:

  ```
  fs.readdir("./public/images/uploaded", function(err, items) …)
  ```

  here, *items* is an array including all the images file names (e.g., 1518186273491.jpg). Then, refer to Week 6 notes/ example code, construct an object using the array (items) as data for images.hbs. Your statement in server.js may look like:

  res.render("images", object containing "images" array here);

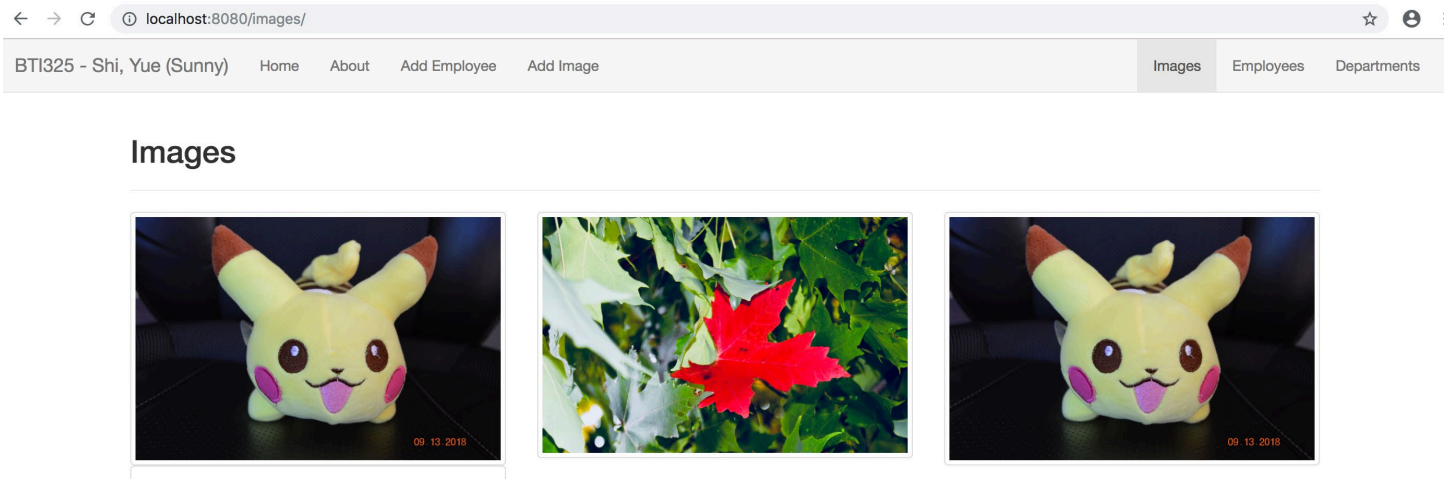
  so that you can send the object containing the array of images as data for your "images" view.

- Once this is complete, modify your images.hbs file using the handlebars #each helper to iterate over the "images" array, such that ***every* image** is shown **in its own ... element** (effectively replacing our single "static" image). This will have the effect of giving us a nice, responsive grid of multiple "col-md-4" columns, each containing its own image.
  - **NOTE:** you can **directly** use **{{this}}** to replace the original static image file name (e.g., 1518186273491.jpg) within the loop to get the current value of the item in the array of strings.

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

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

- Test the route /images, you may see something like this:



- **NOTE**: since we are hosting our app on Heroku, you will notice that once the app "sleeps" and starts up again, any uploaded images 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 images to Cloudinary from your server.js code and store them online using their service)

Part 3: Updating the Employees Route & Adding a View

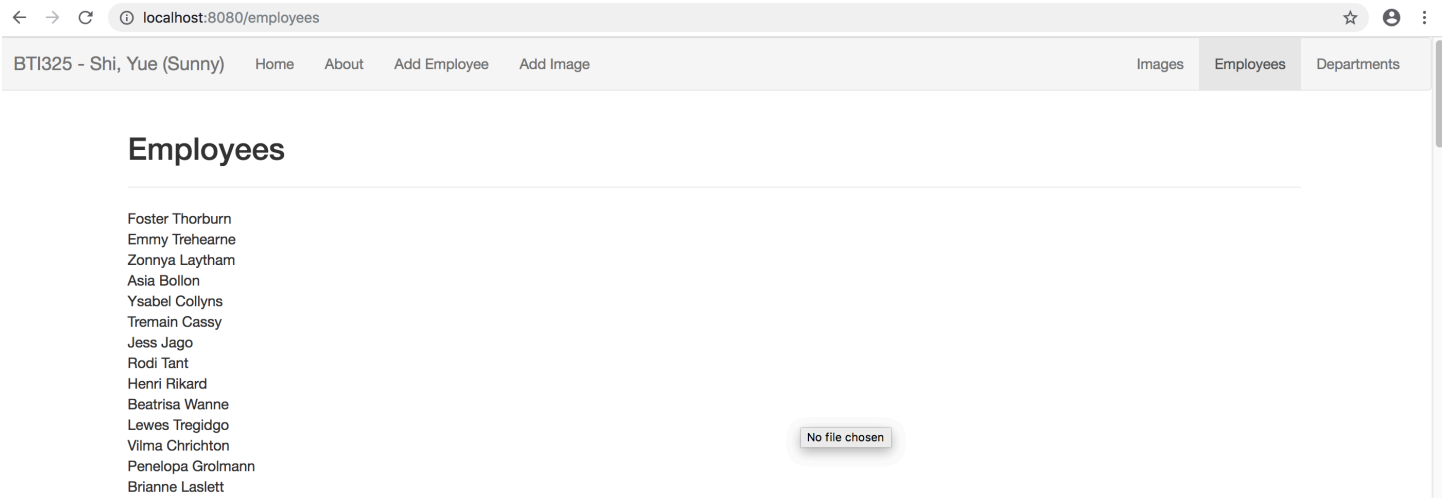
Rather than simply outputting a list of employees using res.json, it would be much better to actually render the data in a table that allows us to access individual employees and filter the list using our existing req.params code.

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

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

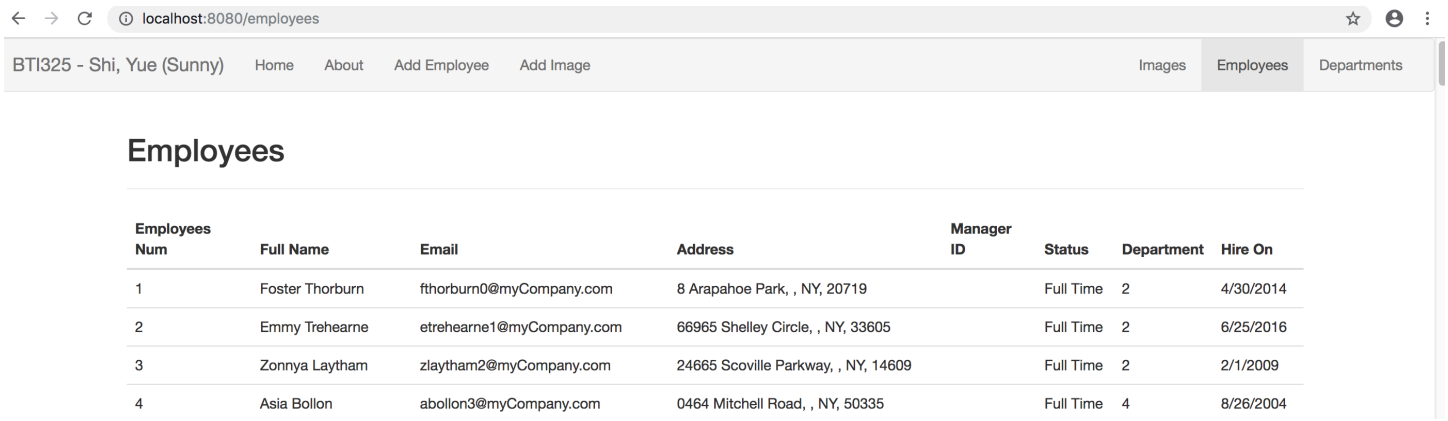
```
<div class="row">
    <div class="col-md-12">
        <h2>Employees</h2>
        <hr />

        <p>TODO: refer to week 6 notes/example code,
                render a list of all employee first and last names here</p>

    </div>
</div>
```

- Replace the <p> element (containing the TODO message) with code to iterate over **each employee** and simply render their first and last names (you may assume that there will be an "employees" array (see below).
- Once this is done, update your GET "/employees" route according to the following specification
    - Every time you would have used res.json(data), modify it to instead use res.render("employees", {employees: 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 "/employees" route:

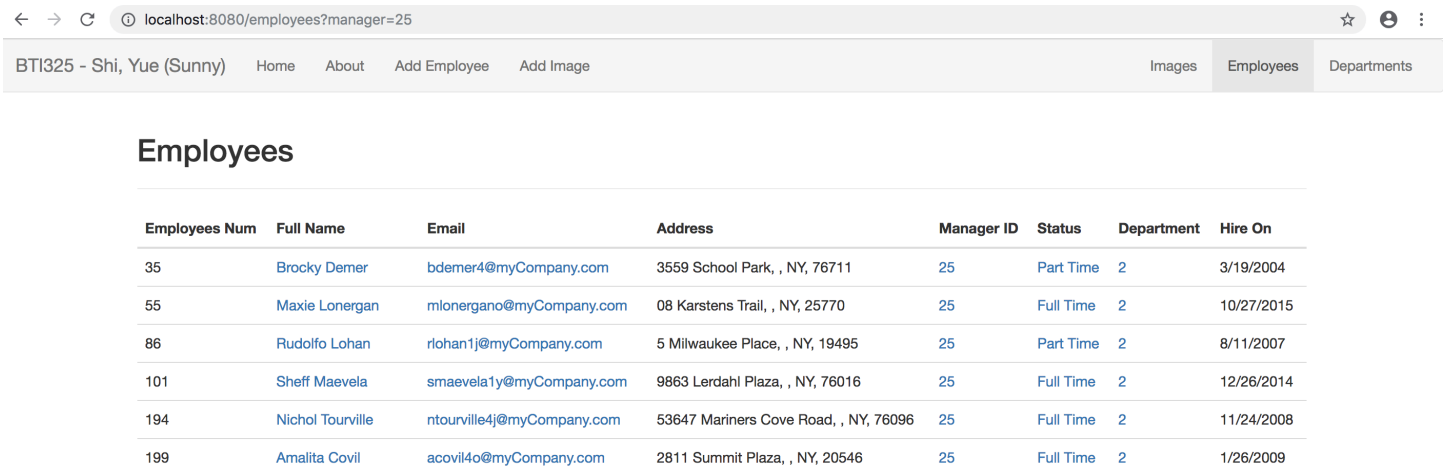


Step 2: Building the Table & Displaying the error "message"

- Update the employees.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 sample here: https://infinite-caverns-60557.herokuapp.com/employees
    - The table must consist of 8 columns with the headings: **Employee Num**, **Full Name**, **Email**, **Address**, **Manager ID**, **Status**, **Department** and **Hired On**
    - Test here, you should see something like:



    - Next, add more work as follows. Test each step before "errors" become more complicated.
    - The Name in the "Full Name" column must link to /employee/**empNum** where **empNum** is the employee number for that row. This route (/employee/**empNum) will be updated below in part 5, step 1.**
    - The "Email" column must be a "mailto" link to the user's email address for that row
    - The "Manager Id" link must link to /employees?manager=**employeeManagerNum** where **employeeManagerNum** is the manager number for the employee for that row
    - The "Status" link must link to /employees?status="Full Time" if "Full Time" is clicked, and /employees?status="Part Time" if "Part Time" is clicked
    - The "Department" link must link to /employees?department=**department** where **department** is the department number for the employee for that row
    - The following is an example screenshot for route /employees?manager=25.



- Add the following code that will conditionally display the "message" if there are no employees. Note, {{message}} is the message from .catch in the route in **server.js**.

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

Part 4: Updating the Departments Route & Adding a View

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

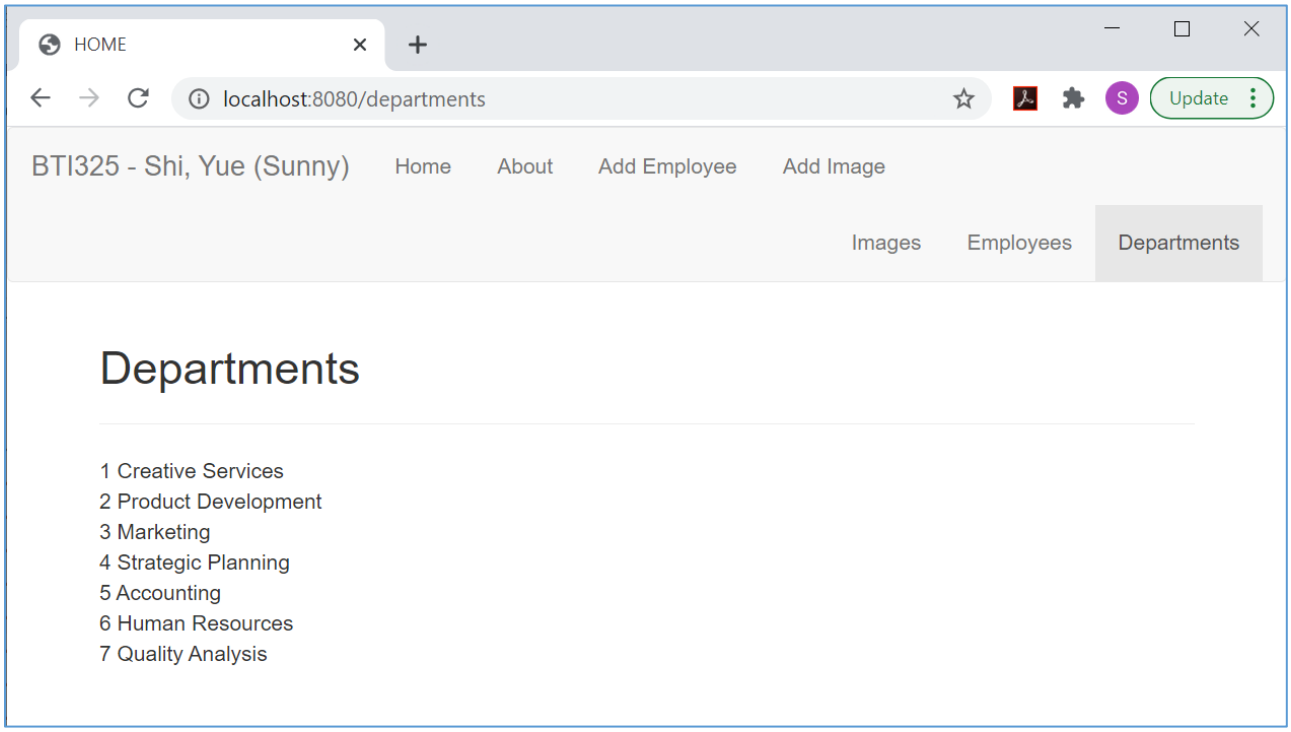
Step 1: Creating a simple "Departments" list & updating server.js

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

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

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

   </div>
</div>
```

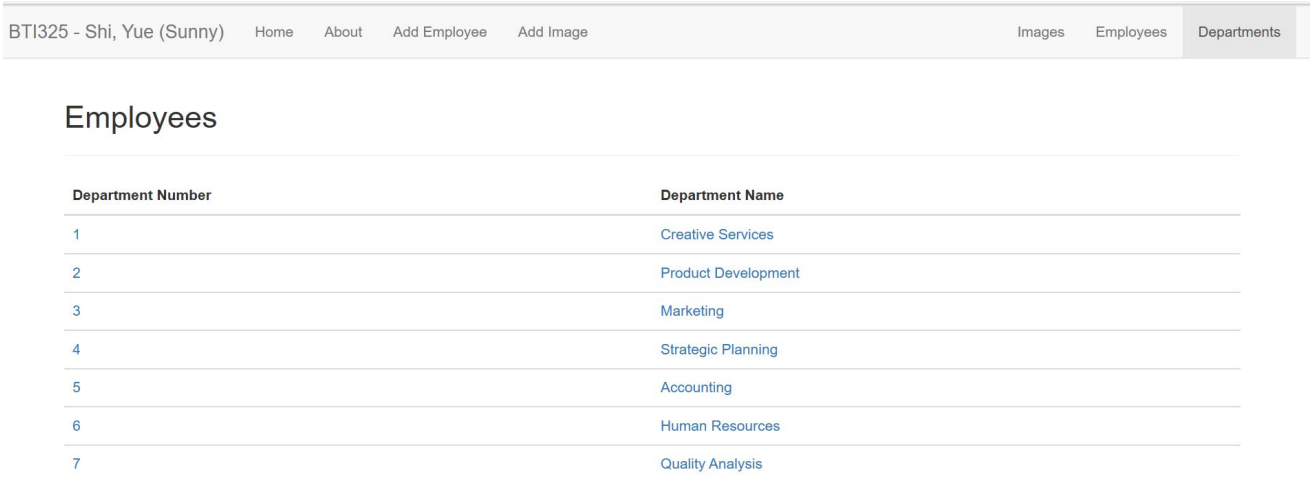
- Replace the <p> element (containing the TODO message) with code to iterate over **each department** and simply render their id and name values (you may assume that there will be a "departments" array (see below).
- Once this is done, update your GET "/departments" route according to the following specification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

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



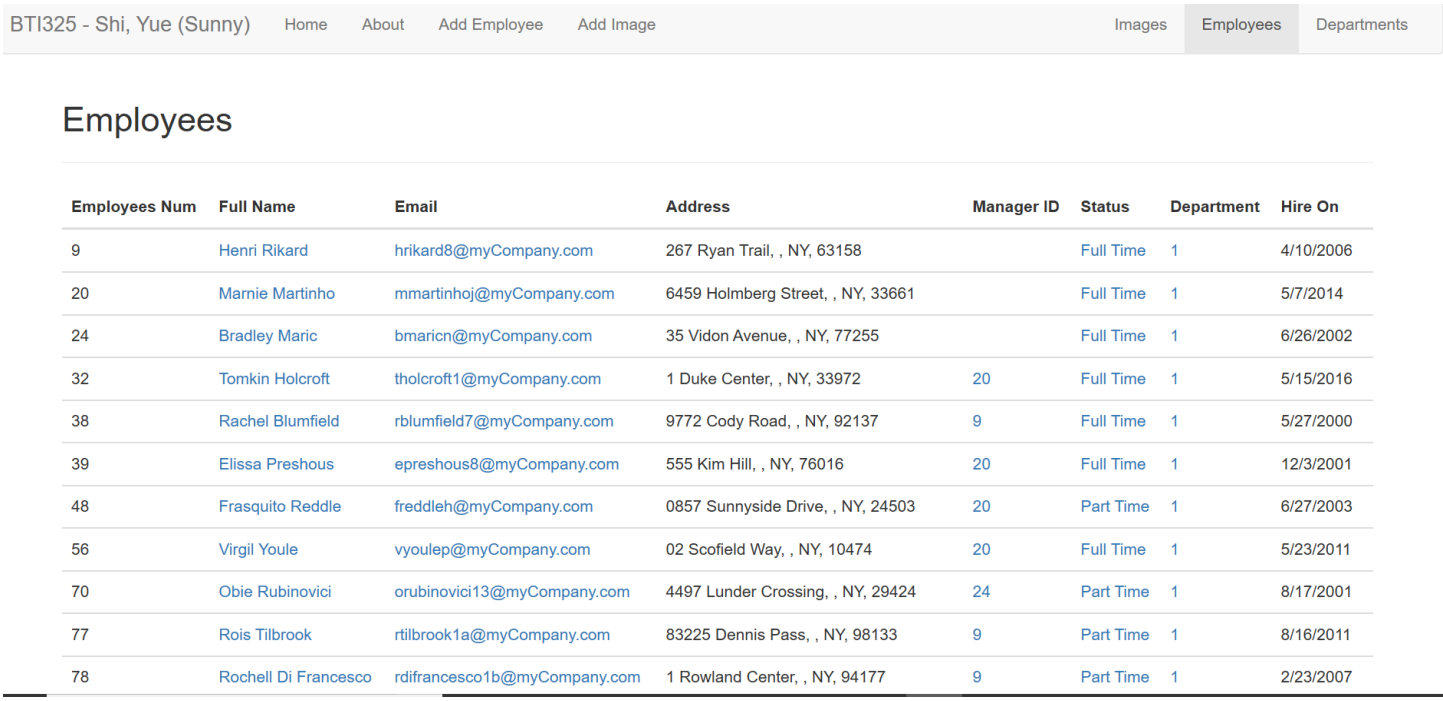
Step 2: Building the Table

- Update the departments.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 2 columns with the headings: **Department Number** and **Department Name**
- Refer to the example screenshot below:



- Note: if you click on either the department id, or the department name, you'll be redirected to /employees?department=X, where X is the department number for the department that was clicked. This route is same as in Part 3, step 2.

  E.g., if click the first department “Creative Services”, it redirected to the following screenshot



Part 5: Updating Existing Employees

The last piece of the assignment is to create a view for a single employee. Currently, when you click on an employee name in the "/employees" route (see above Part 3, Step 2), you will be redirected to a page that shows all of the information for that employee as a JSON-formatted string (ie: accessing **http://localhost:8080/employee/21**, should display a JSON formatted string representing the corresponding employee - employee 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.

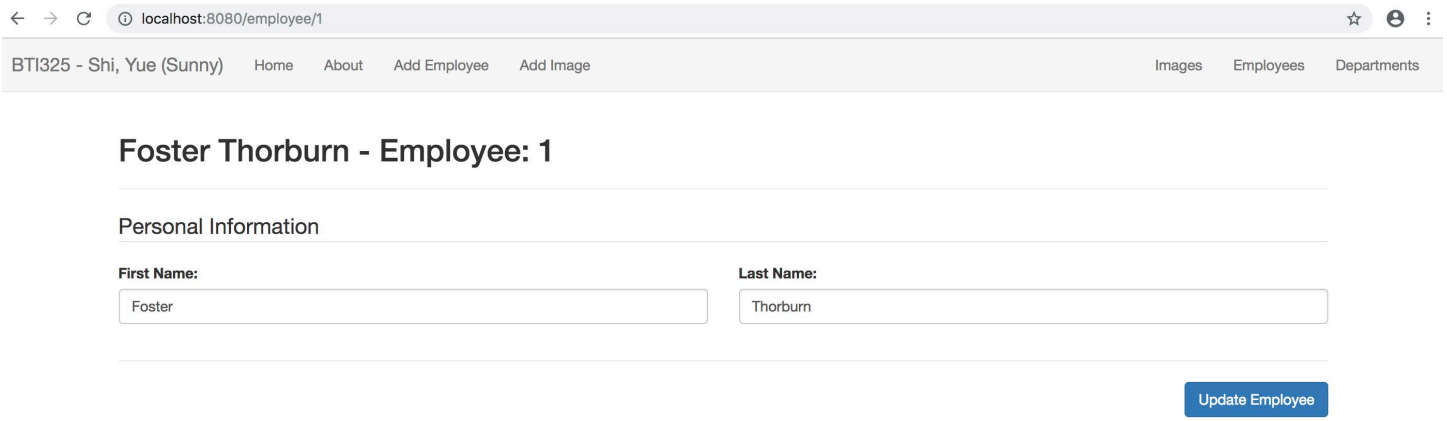
Step 1: Creating new .hbs file / route to Update Employees

- First, add a file "employee.hbs" " in the "views" directory
- Inside the newly created "employee.hbs" view, add the html (**NOTE:** be sure to check that the formatting is correct after pasting the code):

```
<div class="row">
  <div class="col-md-12">
    <h2>{{employee.firstName}} {{ employee.lastName}} - Employee: {{ employee.employeeNum}}</h2>
    <hr />
    <form method="post" action="/employee/update">
      <fieldset>
        <legend>Personal Information</legend>
        <div class="row">
          <div class="col-md-6">
            <div class="form-group">
              <label for="firstName">First Name:</label>
```

```
                    <input class="form-control" id="firstName" name="firstName" type="text"
value="{{ employee.firstName}}" />
                  </div>
                </div>
                <div class="col-md-6">
                  <div class="form-group">
                    <label for="lastName">Last Name:</label>
                    <input class="form-control" id="lastName" name="lastName" type="text"
value="{{ employee.lastName}}" />
                  </div>
                </div>
              </div>
            </fieldset>
            <hr />
            <input type="submit" class="btn btn-primary pull-right" value="Update Employee" /><br /><br /><br />
          </form>
        </div>
      </div>
```

- Once this is done, update your GET "/employee/:empNum" route according to the following specification
    - Use **res.render("employee", { employee: data });** inside the **.then()** callback (instead of res.json) and use **res.render("employee",{message:"no results"});** inside the **.catch()** callback
- Test the server, click “Employees”, then click full name “Foster Thornburn”, **(/employee/1**) . You’ll see:



- Continue this pattern to develop the full form to match the sample:

  https://infinite-caverns-60557.herokuapp.com/employee/1

  Note: you may use the code in the sample to help guide your solution.
    - **employeeNum**: type: "hidden", name: "employeeNum"
    - **Email**: type: "email", name: "email"
    - **Social Security Number:** type: "text", name: "SSN", readonly
    - **Address (Street):** type: "text", name: "addressStreet"
    - **Address (City):** type: "text", name: "addressCity"

- **Address (State):** type: "text", name: "addressState"
- **Address (Zip Code):** type: "text", name: "addressPostal"
- **Manager:** type: "checkbox", name: "isManager", **(HINT**: **use the #if helper - {{#if data.isManager}} … {{/if}} to see if the checkbox should be checked or not)**
- **Employee's Manager Number:** type: "text", name: "employeeManagerNum"
- **Status:** type: "radio" name: "status", values: "Full Time" or "Part Time" **(HINT**, **use the #equals helper - {{#equal data.status "Full Time" }} checked {{/equal}} , to see if Full Time or Part Time is checked)**
- **Department** type: "select", name: "department", values: 1 - 7 inclusive **(HINT**, **use the #equals helper - {{#equal data.department "1" }} selected {{/equal}} for each option to determine which <option> should be selected)**
- **Hire Date** type: "text", name: "hireDate", readonly

- 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: **/employee/update** in our server.js file:

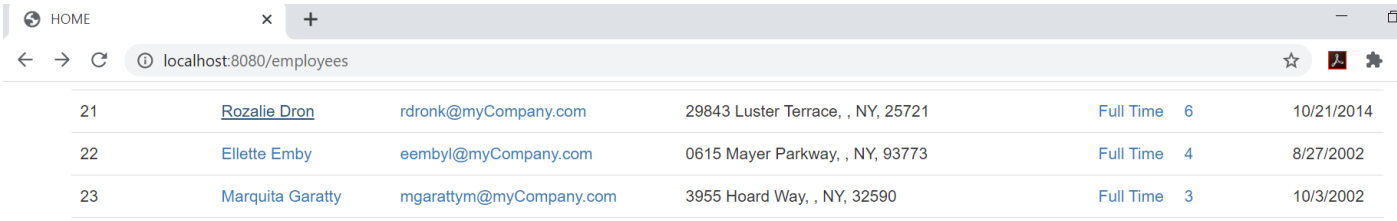
```
app.post("/employee/update", (req, res) => {
  console.log(req.body);
  res.redirect("/employees");
});
```

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

Step 2: Updating the data-service.js module

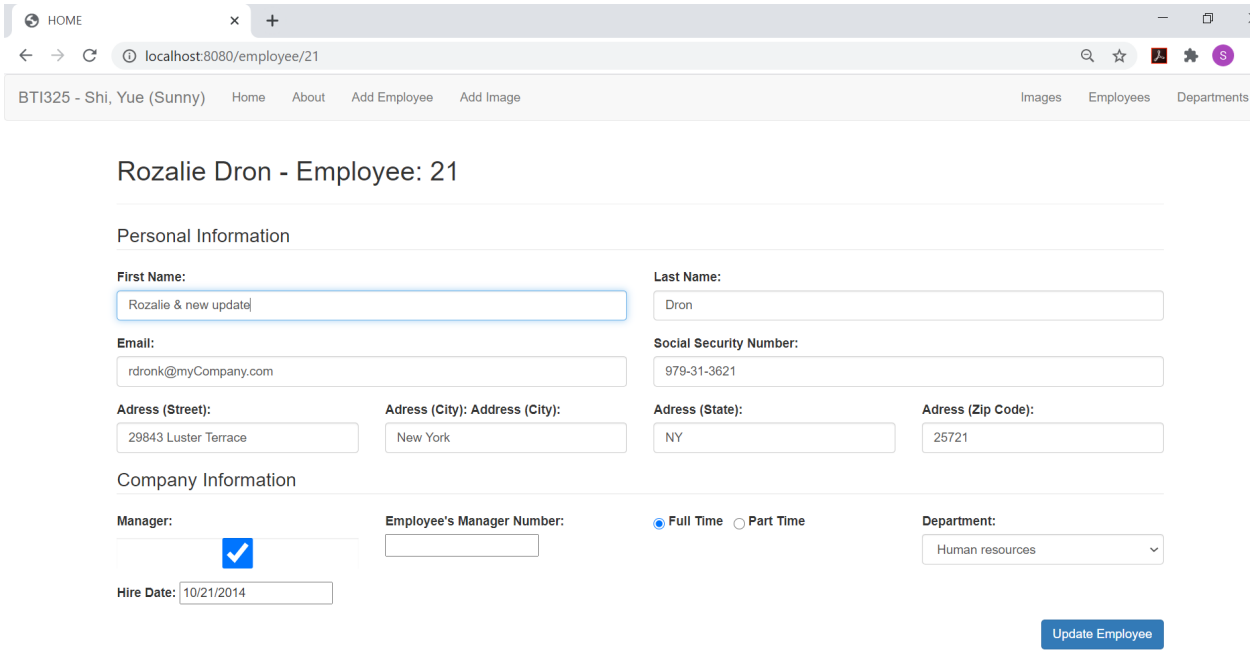
- Add the new method: **updateEmployee(employeeData)** that **returns a promise**. This method will:
  - Search through the "employees" array for an employee with an employeeNum that matches the JavaScript object (parameter employeeData).
  - When the matching employee is found, overwrite it with the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

  (1) In “Employees” page, click Employee 21 (Rozalie Dron)



(2) It directs you to the page of that employee with ID 21. Pay attention to the url, the route: /employee/21

(3) Update the name (or any information), then click button “Update” , the information will be updated.



Part 6: Pushing to Heroku

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

1) Ensure that you have checked in your latest code using **git** (from within Visual Studio Code)
2) Open the integrated terminal in Visual Studio Code
3) Log in to your Heroku account using the command **heroku login**
4) Create a new app on Heroku using the command **heroku create (you can use the same url from your A3, then you can skip this step and go to next step: git push heroku master)**
5) 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 grades for previous work, it is safe to delete those apps (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 4
* 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** -> **A**4

Important Note:

- Compress (.zip) your bti325-app folder and submit the .zip file to My.Seneca under **Assignments** -> **A**4
- 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.