BTI325 Assignment 5

Due: Sunday, Nov 22, 2020 @ 11:59 PM

Objective:

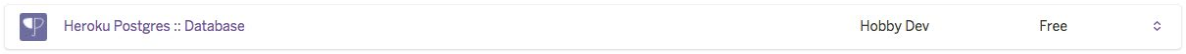
Work with a Postgres data source on the server and practice refactoring an application.

NOTE: If you are unable to start this assignment because Assignment 4 was incomplete - email me for a clean version of Assignment 4 files to start from (effectively removing any custom CSS or text added to your solution).

Getting Started:

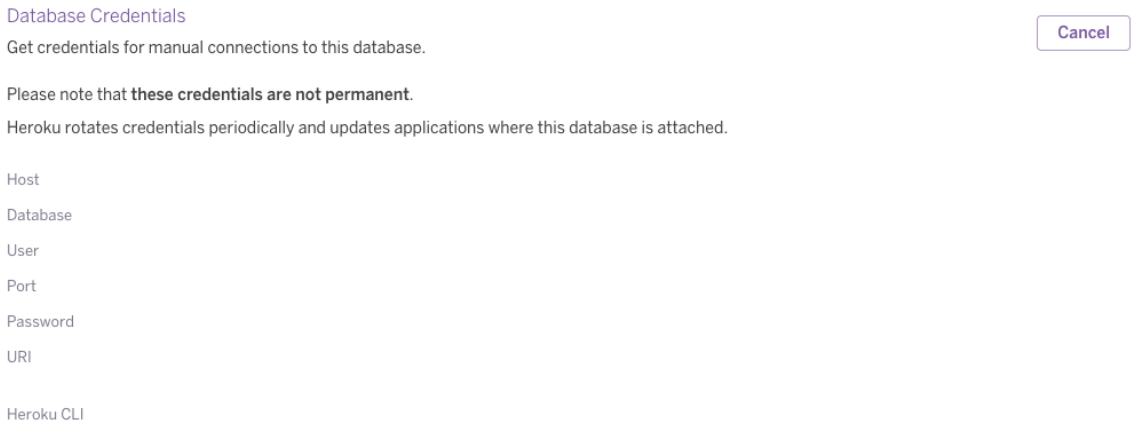
Before we get started, we must add a new Postgres instance on our bti325 app in Heroku:

- Navigate to the application from your Heroku dashboard
- Click the "Resources" header and type "Postgres" in the bottom text box labeled: "Add-ons"
- Click the "Heroku Postgres" link that comes up
- This should cause a modal window to appear. Keep the settings the way they are "Hobby Dev --Free" and click "Provision"
- Click on the new Add-on "Heroku Postgres :: Database" linknpm



- In the new page, click the "Settings" link. Here, you will see an "Administration" section. Click the "View Credentials..." button

- Record all of the credentials (ie: Host, Database, User, Port, etc.) - we will be using them to connect to the database:



- The pre-recorded video is available on Blackboard with step-by-step demo.

Getting Started - Cleaning the solution

- To begin: open your Assignment 4 folder in Visual Studio Code
- In this assignment, we will no longer be reading the files from the "data" folder, so remove this folder from the solution
- Inside your **data-service.js** module, **delete** any code that is **not** a **module.exports** function (ie: global variables, & "require" statements)
- Inside **every single module.exports** function (ie: module.exports.initialize(), module.exports.getAllEmployees, module.exports.getEmployeesByStatus, etc.), remove all of the code and replace it with a return call to an "empty" promise that invokes reject() - (Note: we will be updating these later), ie:

```
return new Promise(function (resolve, reject) {
    reject();
});
```

Installing "sequelize"

- Open the "integrated terminal" in Visual Studio Code and enter the commands to install the following modules (e.g., npm install sequelize):
    - sequelize
    - pg
    - pg-hstore
- At the top of your **data-service.js** module, add the lines:
    - **const** Sequelize = require('sequelize');
    - **var** sequelize = **new** Sequelize('database', 'user', 'password', {
      host: 'host',
      dialect: 'postgres',
      port: 5432,
      dialectOptions: {
      ssl: true
      }
      });

- **NOTE:** for the above code to work, replace *'database'*, *'user'*, *'password'* and *'host'* with the credentials that you saved when creating your new Heroku Postgres Database (above)

- In your **data-service.js** module, add the following lines to test the database connection successful.

```
sequelize.authenticate().then(()=> console.log('Connection success.'))
.catch((err)=>console.log("Unable to connect to DB.", err));
```

You are expected to see the result like:

Executing (default): SELECT 1+1 AS result
Connection success.

Which means the connection to database is successful and you can continue. Otherwise stop and double check your database credentials are input correctly until success.

Creating Data Models

- Inside your **data-service.js** module (before your module.exports functions), define the following 2 data models (**HINT**: See "Models (Tables) Introduction" in “Week 7 Notes”)

- Employee

Column Name	Sequelize DataType
employeeNum	Sequelize.INTEGER primaryKey autoIncrement
firstName	Sequelize.STRING
lastName	Sequelize.STRING
email	Sequelize.STRING
SSN	Sequelize.STRING
addressStreet	Sequelize.STRING
addressCity	Sequelize.STRING
addressState	Sequelize.STRING
addressPostal	Sequelize.STRING
maritalStatus	Sequelize.STRING
isManager	Sequelize.BOOLEAN
employeeManagerNum	Sequelize.INTEGER
status	Sequelize.STRING
department	Sequelize.INTEGER
hireDate	Sequelize.STRING

- Department

Column Name	Sequelize DataType
departmentId	Sequelize.INTEGER primaryKey autoIncrement
departmentName	Sequelize.STRING

Update Existing data-service.js functions

Now that we have Sequelize set up properly, and our "Employee" and "Department" models defined, we can use all of the Sequelize operations, discussed in “Week 7 Notes” to update our data-service.js to work with the database:

initialize()

- This function will invoke the sequelize.sync() function, which will ensure that we can connected to the DB and that our Employee and Department models are represented in the database as tables.
- If the **sync()** operation resolved **successfully**, invoke the **resolve** method for the promise (no data passed) 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 sync the database").

getAllEmployees()

- This function will invoke the Employee.findAll() function
- If the **Employee.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data) to communicate back to server.js that the operation was a success and to provide the data.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: "no results returned".

getEmployeesByStatus(***status***)

- This function will invoke the Employee.findAll() function and filter the results by "status" (using the value passed to the function - ie: "Full Time" or "Part Time"
- If the **Employee.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data) to communicate back to server.js that the operation was a success and to provide the data.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: "no results returned".

getEmployeesByDepartment(***department***)

- This function will invoke the Employee.findAll() function and filter the results by "department" (using the value passed to the function - ie: 1 or 2 or 3 … etc

- If the **Employee.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data) to communicate back to server.js that the operation was a success and to provide the data.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: "no results returned".

getEmployeesByManager(***manager***)

- This function will invoke the Employee.findAll() function and filter the results by "employeeManagerNum" (using the value passed to the function - ie: 1 or 2 or 3 … etc
- If the **Employee.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data) to communicate back to server.js that the operation was a success and to provide the data.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: "no results returned".

getEmployeeByNum(***num***)

- This function will invoke the Employee.findAll() function and filter the results by "employeeNum" (using the value passed to the function - ie: 1 or 2 or 3 … etc
- If the **Employee.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data[0], ie: only provide the first object) to communicate back to server.js that the operation was a success and to provide the data.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: "no results returned".

getDepartments()

- This function will invoke the Department.findAll() function
- If the **Department.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data) to communicate back to server.js that the operation was a success and to provide the data.
- If there was an error at any time during this process (or no results were returned), invoke the **reject** method and pass a meaningful message, ie: "no results returned".

addEmployee(***employeeData***)

- Before we can work with ***employeeData*** correctly, we must once again make sure the isManager property is set properly. Recall: to ensure that this value is set correctly, before you start working with the employeeData object, add the line:
    - **employeeData.isManager = (employeeData.isManager) ? true : false;**
- Additionally, we must ensure that any blank values ("") for properties are set to null. For example, if the user didn't enter a Manager Number (causing employeeData.employeeManagerNum to be ""), this needs to be set instead to null (ie: employeeData.employeeManagerNum = null). You can iterate over every property in an object (to check for empty values and replace them with null) using a for…in loop.

- Now that the isManager is explicitly set (true or false), and all of the remaining "" are replaced with null, we can invoke the Employee.create() function
- If the **Employee.create()** operation resolved **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 and pass a meaningful message, ie: "unable to create employee".

updateEmployee(***employeeData***)

- Like addEmployee(employeeData) we must ensure that the ***isManager*** value is explicitly set to true/false and any blank values in ***employeeData*** are set to null (follow the same procedure)
- Now that all of the "" are replaced with null, we can invoke the Employee.update() function and filter the operation by "employeeNum" (ie employeeData.employeeNum)
- If the **Employee.update()** operation resolved **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 and pass a meaningful message, ie: "unable to update employee".

Updating the Navbar & Existing views (.hbs)

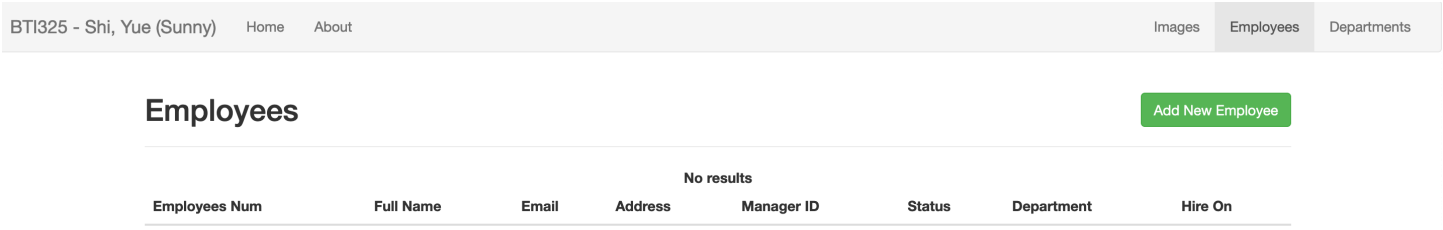
If we test the server now and simply navigate between the pages, we will see that everything still works, except we no longer have any employees in our "Employees" view, and no departments within our "Departments" view. This is to be expected (since there is nothing in the database), however we are not seeing an any messages (just empty tables). To solve this, we must update our server.js file:

- /employees route
    - Where we would normally render the "employees" view with data
        - ie: res.render("employees", {employees:data});

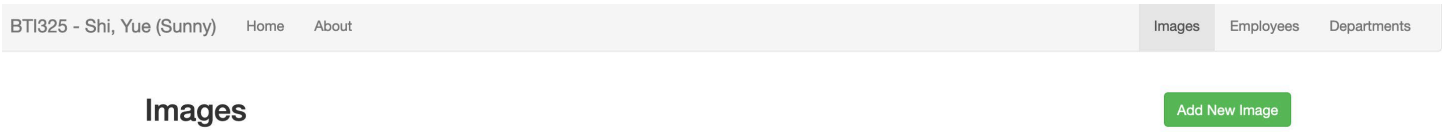
      we must place a condition there first so that it will only render "employees" if data.length > 0. Otherwise, render the page with an error message,
        - ie: res.render("employees",{ message: "no results" });
    - If we test the server now, we should see our "no results" message in the /employees route
    - **NOTE**: We must still show error messages if the promise(s) are rejected, as before
- /departments route
    - Using the same logic as above (for the /employees route) update the /departments route as well
    - If we test the server now, we should see our "no results" message in the /departments route
    - **NOTE**: We must still show an error message if the promise is rejected, as before

For this assignment, we will be moving the "add Employee" and "add Image" links into their respective pages (ie: "add Employee" will be moved out of the Navbar and into the "employees" view and "add Image" will be moved out of the Navbar and into the "images" view)

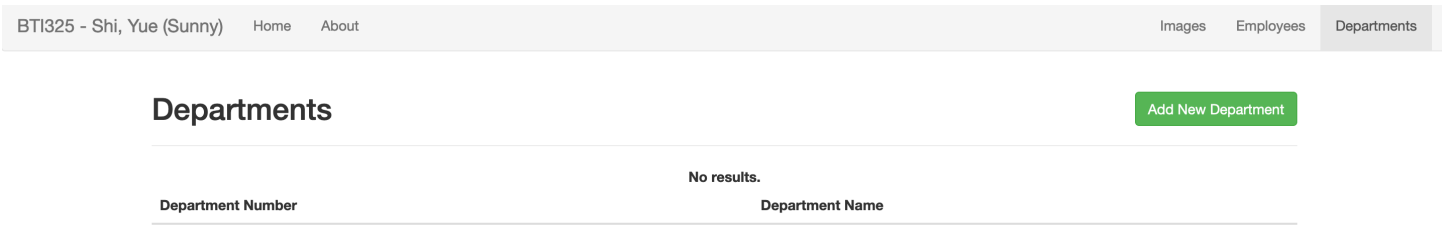
- Link "add Employee" (refer to screenshot below)
    - Remove this link ({{#navLink}} … {{/navLink}}) from the "navbar-nav" element inside the main.hbs file
    - Inside the "employees.hbs" view (Inside the <h2>Employees</h2> element), add the below code to create a "button" that links to the "/employees/add" route:
        - Add New Employee



- Link "add Image" (refer to screenshot below)
    - Remove this link ({{#navLink}} … {{/navLink}}) from the "navbar-nav" element inside the main.hbs file
    - Inside the "images.hbs" view (Inside the <h2>Images</h2> element), add the below code to create a "button" that links to the "/images/add" route:
        - Add New Image



- "add Department" (refer to screenshot below)
    - You will notice that currently, we have no way of adding a new department. However, while we're adding our "add" buttons, it makes sense to create an "add Department" button as well (we'll code the route and data service later in this assignment).
    - Inside the "departments.hbs" view (Inside the <h2>Departments</h2> element), add the below code to create a "button" that links to the "/departments/add" route:
        - Add New Department



Adding new data-service.js functions

So far, all our data-service functions have focused primarily on fetching data and only adding/updating Employee data. If we want to allow our users to fully manipulate the data, we must add some additional promise-based data-service functions to add/update Departments:

addDepartment(***departmentData***)

- Like addEmployee(employeeData) function we must ensure that any blank values in ***departmentData*** are set to null (follow the same procedure)
- Now that all of the "" are replaced with null, we can invoke the Department.create() function
- If the **Department.create()** operation resolved **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 and pass a meaningful message, ie: "unable to create department"

updateDepartment(***departmentData***)

- Like addDepartment(departmentData) function we must ensure that any blank values in ***departmentData*** are set to null (follow the same procedure)
- Now that all of the "" are replaced with null, we can invoke the Department.update() function and filter the operation by "departmentId" (ie departmentData.departmentId)
- If the **Department.update()** operation resolved **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 and pass a meaningful message, ie: "unable to update department".

getDepartmentById(***id***)

- Similar to the getEmployeeByNum(***num***) function, this function will invoke the Department.findAll() function (instead of Employee.findAll()) and filter the results by "id" (using the value passed to the function - ie: 1 or 2 or 3 … etc
- If the **Department.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data[0], ie: only provide the first object) to communicate back to server.js that the operation was a success and to provide the data.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: "no results returned".

Updating Routes (server.js) to Add / Update Departments

Now that we have our data-service up to date to deal with department data, we need to update our server.js file to expose a few new routes that provide a form for the user to enter data (GET) and for the server to receive data (POST).

/departments/add

- This **GET** route is very similar to your current "/employees/add" route - only instead of "rendering" the "addEmployee" view, we will instead set up the route to "render" an "addDepartment" view (added later)

/departments/add

- This **POST** route is very similar to your current "/employees/add" POST route - only instead of calling the addEmployee() data-service function, you will instead call your newly created addDepartment() function with the POST data.
- Instead of redirecting to /employees when the promise has resolved (using .then()), we will instead redirect to /departments

/department/update

- This **POST** route is very similar to your current "/employee/update" route - only instead of calling the updateEmployee() data-service function, you will instead call your newly created updateDepartment() function with the POST data.
- Instead of redirecting to /employees when the promise has resolved (using .then()), we will instead redirect to /departments

/department/:departmentId

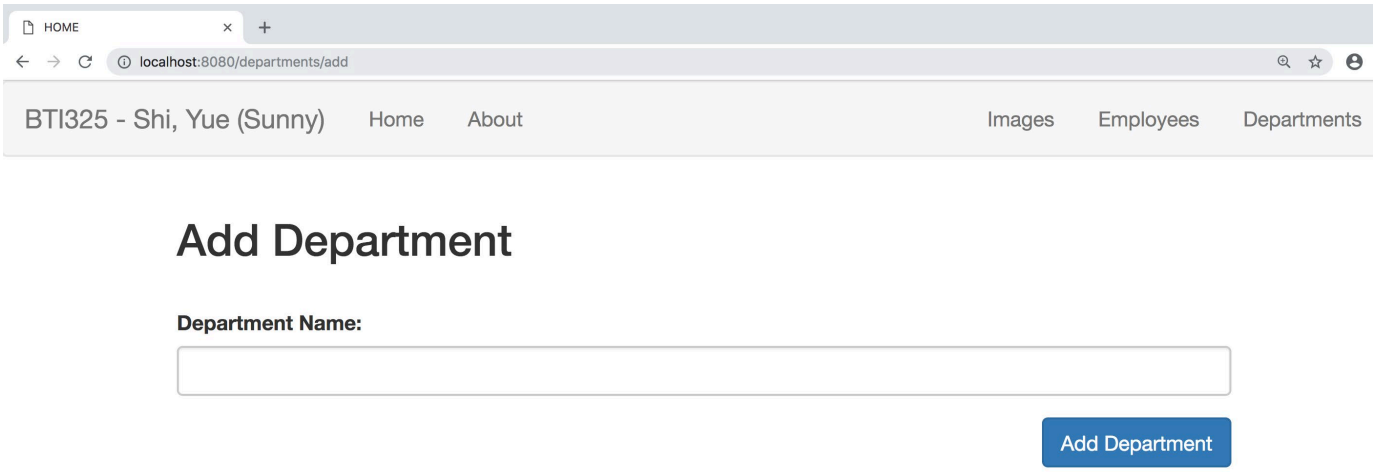
- This **GET** route is very similar to your current "/employee/:empNum" route - only instead of calling the getEmployeeByNum() data-service function, you will instead call your newly created getDepartmentById() function with the departmentId parameter value.
- Once the **getDepartmentById(id)** operation has resolved, we **then** "render" a "department" view (added later) and pass in the data from the promise. If the data is undefined (ie, no error occurred, but no results were returned either), send a 404 error back to the client using: **res.status(404).send("Department Not Found");**
- If the getDepartmentById(id) promise is rejected (using .**catch**), send a 404 error back to the client using: **res.status(404).send("Department Not Found");**

Updating Views to Add / Update Departments

In order to provide user interfaces to all of our new "Department" functionality, we need to add / modify some views within the "views" directory of our app:

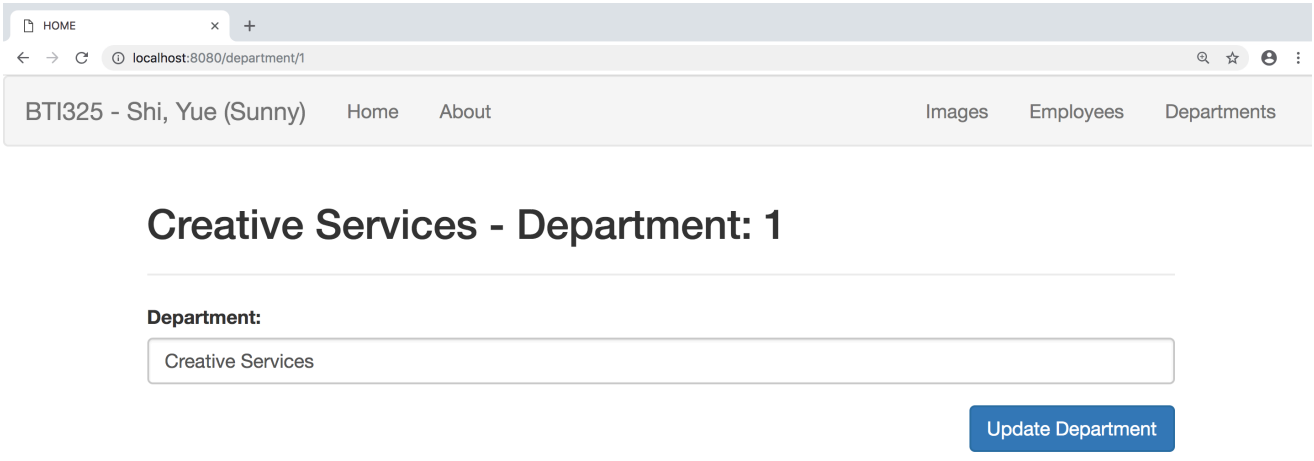
addDepartment.hbs

- Fundamentally, this view is nearly identical to the **addEmployee.hbs** view, however there are a few key changes:
    - The header (<h2>…</h2>) must read "Add Department"
    - The form must submit to "/departments/add"
    - There must be only one input field (type: "text", name: "departmentName", label: "Department Name")
    - The submit button must read "Add Department"
- When complete, your view should appear as:



department.hbs

- Like addDepartment.hbs, the department.hbs view is very similar to its employee counterpart: employee.hbs, only with a few key differences:
    - The header (<h2>…</h2>) must read "***departmentName*** - Department: ***departmentId***" where ***departmentName*** and ***departmentId*** represent the departmentName and departmentId of the current department.
    - The form must submit to "/departments/update"
    - The **hidden field** must have the properties: name="departmentId" and value="{{data.departmentId}}" in order to keep track of exactly which department the user is editing
    - There must be only one visible input field (type: "text", name: "departmentName", label: "Department Name") its value must be preset to the current department's ***departmentName*** (hint: value="{{data.departmentName}}")
    - The submit button must read "Update Department"
    - When complete, your view should appear as the following (if we're currently looking at a newly created "Creative Services", for example):



departments.hbs

- Lastly, to enable users to access all of this new functionality, we need to make one important change to our current departments.hbs file:
    - **Replace** the code:
      <td>{{departmentName}}</td>
      with
      <td>{{departmentName}}</td>

Now, users can click on the department name if they wish to edit an existing department, or the department number, if they wish to filter the employee list by department number!

Updating the "Department" List in the Employee Views

Now that users can add new Departments, it makes sense that all of the Departments available to an employee (either when adding a new employee or editing an existing one), should consist of all the current departments in the database (instead of just 1…7). To support this new functionality, we must make a few key changes to our routes and views:

"/employees/add" route

- Since the "addEmployee" view will now be working with actual Departments, we need to update the route to make a call to our data-service module to "getDepartments".
- Once the **getDepartments()** operation has resolved, we **then** "render" the " addEmployee view (as before), however this time we will pass in the data from the promise, as "departments", ie: **res.render("addEmployee", {departments: data});**
- If the getDepartments() promise is rejected (using .**catch**), "render" the " addEmployee view anyway (as before), however instead of sending the data from the promise, send an empty array for "departments, ie: **res.render("addEmployee", {departments: []});**

"addEmployee.hbs" view

- Update the: **<select class="form-control" name="department" id="department">**…**</select>** element to use the new handlebars code:

```
{{#if departments}}
    <select class="form-control" name="department" id="department">
        {{#each departments}}
            <option value="{{departmentId}}">{{departmentName}}</option>
        {{/each}}
    </select>
{{else}}
    <div>No Departments</div>
{{/if}}
```

- Now, if we have any departments in the system, they will show up in our view - otherwise we will show a div element that states "No Departments"

"/employee/:empNum" route

- If we want to do the same thing for existing employees, the task is more complicated: In addition to sending the **current Employee** to the "employee" view, we must also send all of the **Departments**. This requires two separate calls to our data-service module ("dataService" in the below code) that returns data that needs to be sent to the view. Not only that, but we must ensure that the right department is selected for the employee and respond to any errors that might occur during the operations. To ensure that this all works correctly, use the following code for the route:

```
app.get("/employee/:empNum", (req, res) => {

  // initialize an empty object to store the values
  let viewData = {};

  dataService.getEmployeeByNum(req.params.empNum).then((data) => {
    if (data) {
      viewData.employee = data; //store employee data in the "viewData" object as "employee"
    } else {
      viewData.employee = null; // set employee to null if none were returned
    }
  }).catch(() => {
    viewData.employee = null; // set employee to null if there was an error
  }).then(dataService.getDepartments)
  .then((data) => {
    viewData.departments = data; // store department data in the "viewData" object as "departments"

    // loop through viewData.departments and once we have found the departmentId that matches
    // the employee's "department" value, add a "selected" property to the matching
    // viewData.departments object

    for (let i = 0; i < viewData.departments.length; i++) {
      if (viewData.departments[i].departmentId == viewData.employee.department) {
        viewData.departments[i].selected = true;
      }
    }

  }).catch(() => {
    viewData.departments = []; // set departments to empty if there was an error
  }).then(() => {
    if (viewData.employee == null) { // if no employee - return an error
      res.status(404).send("Employee Not Found");
    } else {
      res.render("employee", { viewData: viewData }); // render the "employee" view
    }
  });
});
```

"employee.hbs" view

- Now that we have all of the data for the employee inside "viewData.employee" (instead of just "employee"), we must update every handlebars reference to data, from ***employee.propertyName*** to

viewData.employee.propertyName. For example: **{{employee.firstName}}** would become: **{{viewData.employee.firstName}}**

- Once this is complete, we need to update the **<select class="form-control" name="department" id="department">…</select>** element as well:

```
{{#if viewData.departments}}
    <select class="form-control" name="department" id="department">
    {{#each viewData.departments}}
        <option value="{{departmentId}}" {{#if selected}} selected {{/if}} >{{departmentName}} </option>
    {{/each}}
    </select>
{{else}}
    <div>No Departments</div>
{{/if}}
```

Updating server.js, data-service.js & employees.hbs to Delete Employees

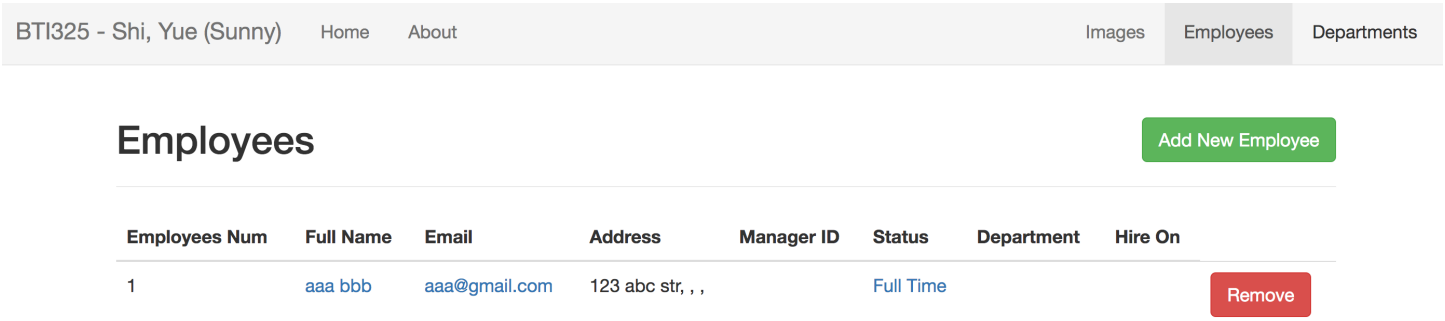
To make the user-interface more usable, we should allow users to remove (delete) employees that they no longer wish to be in the system. This will involve:

- Creating a new function (ie: **deleteEmployeeByNum(empNum)**) in data-service.js to "**delete**" employees using the Employee.destroy() for a specific employee. Ensure that this function returns a **promise** and only "resolves" if the Employee was deleted ("destroyed"). "Reject" the promise if the "destroy" method encountered an error (was rejected).

- Create a new GET route (ie: "**/employees/delete/:empNum**") that will invoke your newly created **deleteEmployeeByNum(empNum)** data-service method. If the function resolved successfully, redirect the user to the **"/employees"** view. If the operation encountered an error, return a **status code of 500** and the plain text: **"Unable to Remove Employee / Employee not found)"**

- Lastly, update the **employees.hbs** view to include a "remove" link for every employee within in a new column of the table (at the end) - Note: The header for the column should not contain any text. The links in every row should be styled as a button (ie: **class="btn btn-danger"**) with the text "**Remove**" and link to the newly created GET route "**/employees/delete/*empNum***" where ***empNum*** is the employee number of the employee in the current row. Once this button is clicked, the employee will be deleted and the user will be redirected back to the **"/employees**" list.

- When complete, your view should appear as:



Handling Rejected Promises in server.js

As a final step to make sure our server doesn't unexpectedly stall or crash on users if a promise is rejected, we ***must*** match every .then() callback function with a .catch() callback. For example, if we try to add or update an

employee with a string value for "Employee's Manager Number", Sequelize will throw an "invalid input syntax for integer" error and our route will hang. To resolve this, we must:

- Ensure that whenever we specify a .then() callback, we specify a matching .catch() callback.
- If the behavior isn't defined (ie: redirect or display a specific message), simply return a **status code of 500** and send a string describing the problem, within the .catch() callback function ie:

```
.catch((err)=>{
    res.status(500).send("Unable to Update Employee");
});
```

NOTE: This is a catch-all that we can use for now, however a better solution would be to implement client-side validation with more sophisticated server-side validation, and save this as a last resort.

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 5
* 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** -> **Assignment** 5

Important Note:

- Compress (.zip) your bti325-app folder and submit the .zip file to My.Seneca under **Assignments** -> **A**5
- 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.