WEB700 Assignment 5

Submission Deadline:

Thursday, July 25rd, 2024 @ 11:59 PM

Assessment Weight:

9% of your final course Grade

Objective:

Build upon the code created in Assignment 4 by incorporating the Handlebars view engine to render our JSON data visually in the browser using**. hbs** views and layouts. Additionally, update our collegeData module to allow for courses to be viewed individually using a (read only) web form.

NOTE: If you are unable to start this assignment because Assignment 4 was incomplete - email your professor for a clean version of the Assignment 4 files to start from. Please note: the "home", "about" and "htmlDemo" html files will not be included in the clean version of Assignment 4

Specification:

As mentioned above, this assignment will build upon your code from Assignment 4. To begin, make a copy of your assignment 4 folder and open it in Visual Studio Code. Note: this will copy your .git folder as well (including the "heroku" remote for assignment 4). 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.

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.engine({ … }) and the "extname" property as ".hbs" and the "defaultLayout" property as "main" (See the Week 9 Notes)
    - call app.set() to specify the 'view engine' (See the Week 9 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 theme.css link looks like this href="css/theme.css", it must be *modified* to use a leading "/", ie href="/css/theme.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)
- 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", "addStudent" and "htmlDemo" 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", "/htmlDemo" or "/students/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.path.substring(1);
  app.locals.activeRoute = "/" + (isNaN(route.split('/')[1]) ? route.replace(/\/(?!.*)/, "") : route.replace(/\/(.*)/, ""));
  next();
});
```

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

- Next, we must use the following handlebars custom "helper" (See the Week 9 notes for adding custom "helpers")"

```
navLink: function(url, options){
  return '<li' +
    ((url == app.locals.activeRoute) ? ' class="nav-item active" ' : ' class="nav-item" ') +
    '><a class="nav-link" href="' + url + '">' + options.fn(this) + '</a></li>';
}
```

- This basically allows us to replace all of our existing navbar links, ie: <li class="nav-item">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 {{#equal "a" "a"}} … {{/equal}} 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:
    - <li class="nav-item">About will become {{#navLink "/about"}}About{{/navLink}}
    - **NOTE**: You can remove the "/tas" menu item from main.hbs and the "/tas" route from server.js, as well as the "getTAs()" function from collegeData.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: Updating the Students Route & Adding a View

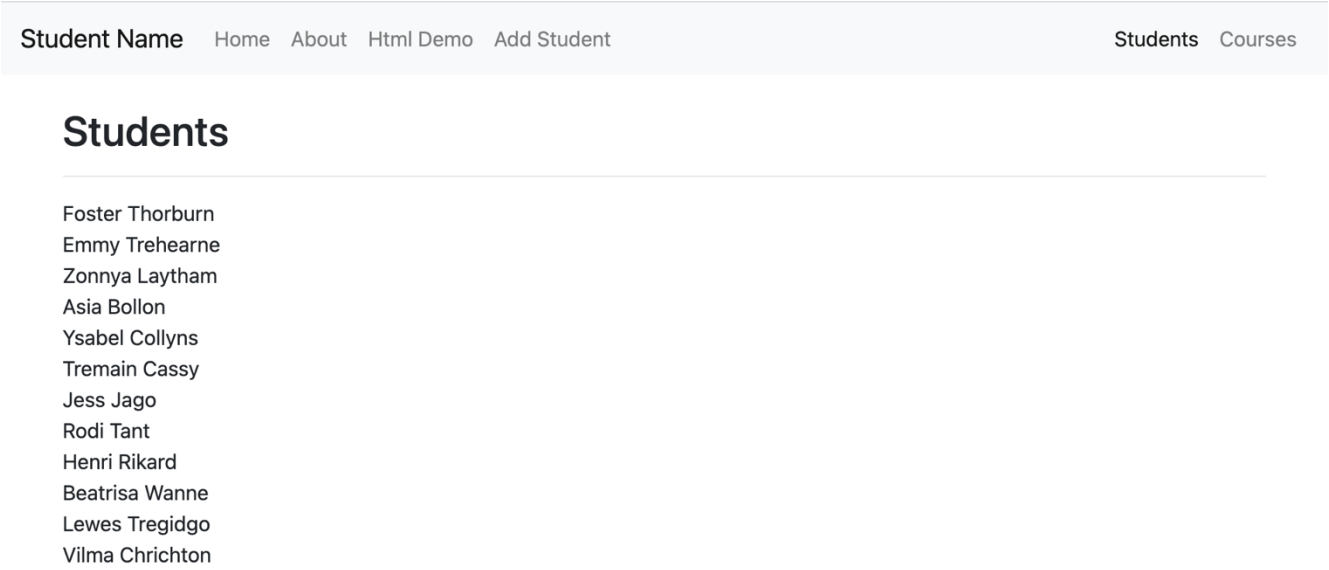
Rather than simply outputting a list of students using res.json, it would be much better to actually render the data in a table that allows us to access individual students and filter the list using our existing req.params code.

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

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

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

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

  </div>
</div>
```

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



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

- Update the students.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://web700-a5-example.herokuapp.com/students
    - The table must consist of 6 columns with the headings: **Student Num**, **Full Name**, **Email**, **Address**, **Status** and **Course ID**
    - Additionally, the Name in the Full Name column must link to /student/**studentNum** where **studentNum** is the student number for that row
    - The "Email" column must be a "mailto" link to the user's email address for that row
    - The "Course" link must link to /students?course=**course** where **course** is the Course ID for the student for that row
- Beneath … element, add the following code that will conditionally display the "message" only if there are no students (**HINT**: #unless students)

```
<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

Part 3: Updating Courses Routes & Adding Views

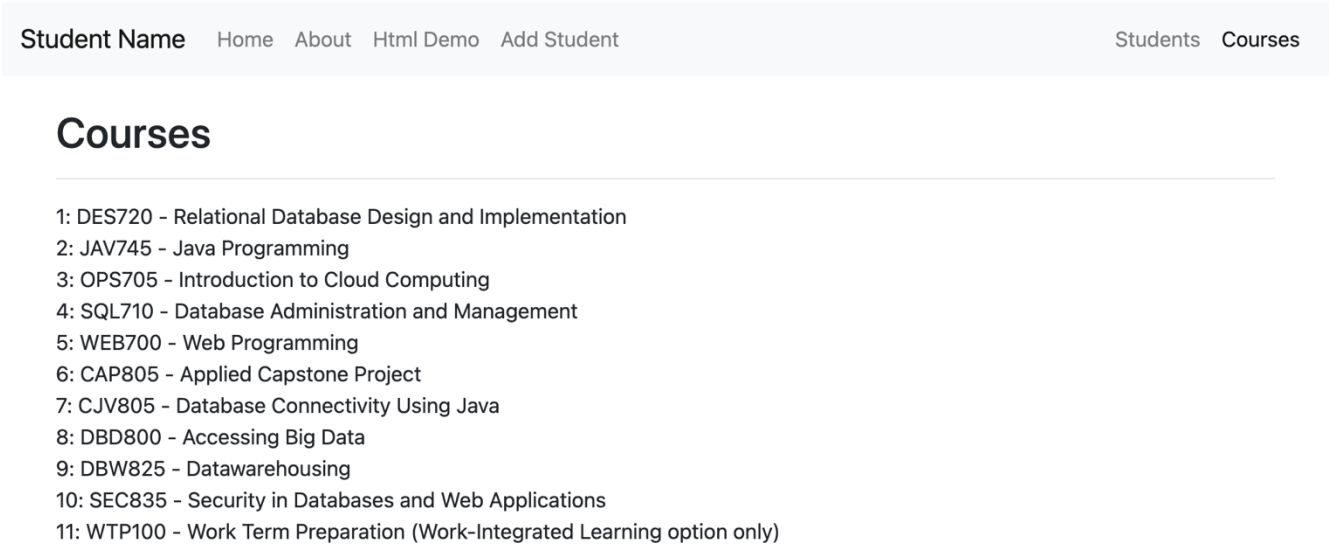
Now that we have the "Student" data rendering correctly in the browser, we can use the same pattern to render the "Courses" data in a table:

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

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

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

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

  </div>
</div>
```

- Replace the <p> element (containing the TODO message) with code to iterate over **each course** and simply render their courseId, courseCode and courseDescription values followed by a
 element (you may assume that there will be a "courses" array (see below).
- Once this is done, update your GET "/courses" route according to the following specification
  - Instead of using res.json(data), modify it to instead use res.render("courses", {courses: 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("courses", {message: "no results"});
- Test the Server - you should see the following page for the "/courses" route:



Step 2: Building the Courses Table

- Update the courses.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: **Course Code** and **Course Description**
- The value in the **Course Code** column must link to /course/**id** where **id** is the **courseId** for that row
- The text in the **Course Description**, column must link to /students?course=**X**, where **X** is the **courseId** for the course that was clicked (see below link for example)
- Refer to the example online at https://web700-a5-example.herokuapp.com/courses
- Beneath … element, add the following code that will conditionally display the "message" only if there are no courses (**HINT**: #unless courses)

```
<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: Creating the Course View & updating server.js / collegeData.js

- First, add a file "course.hbs" in the "views" directory
- Inside the newly created "course.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">
    <br>
    <h2>{{course.courseDescription}}</h2>
    <hr />
    <form>
      <div class="row">
        <div class="col-md-3">
          <div class="form-group">
            <label for="courseDescription">Course Code</label>
            <input class="form-control" id="courseCode" name="courseCode" type="text" readonly
value="{{course.courseCode}}" />
          </div>
        </div>
        <div class="col-md-9">
          <div class="form-group">
            <label for="courseDescription">Course Description</label>
            <input class="form-control" id="courseDescription" name="courseDescription" type="text"
readonly value="{{course.courseDescription}}" />
```

```
            </div>
          </div>
        </div>
        <hr />
      </form>
    </div>
  </div>
```

- Now that we have the form in place, we simply need to create a new "GET" route on our server.js file to "render" the (read only) form for the correct course, as well as a collegeData module function to "getCourseById".
    - First, add a method in your collegeData.js file called "getCourseById".
        - This method takes one parameter (**id**) and returns a **promise**. This function behaves almost exactly like getStudentByNum except for **courses**, ie: the purpose of this function is to simply search the "courses" array and "resolve" the promise with a single course whose **courseId** property matches the **id** parameter.
        - If a course cannot be found, the promise will reject with an appropriate message, ie "query returned 0 results"
    - Next, create a new GET route in the server.js file that will match routes with the pattern: /course/**id** where **id** is the requested course id, ie: "course/4".
        - The purpose of this route is to invoke the newly created "getCourseById" method from the collegeData module with the **id** value sent in the route, in order to get a specific **course** object (functioning almost exactly like the "/student/:studentNum" route).
        - If this function resolves successfully, **render** our new "course" view using the code: **res.render("course", { course: data });** (assuming that "data" is what was resolved from the promise).
- Test the newly created **courses** / **course** views – your solution should behave like the online example here: https://web700-a5-example.herokuapp.com/courses

Part 4: Updating Existing Students

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

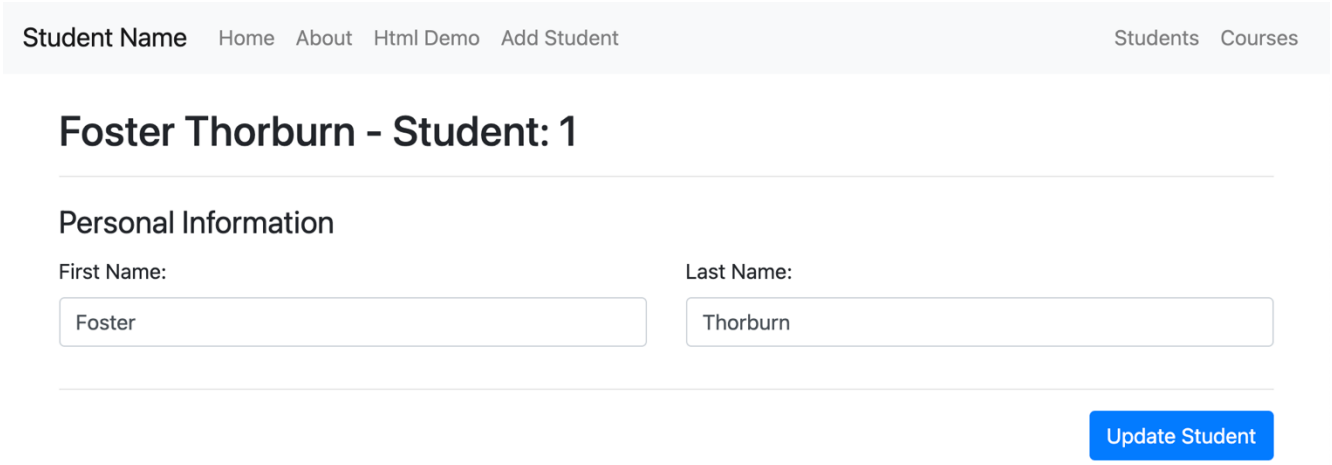
- First, add a file "student.hbs" in the "views" directory

- Inside the newly created "student.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">
  <main class="col-md-12">
    <br>
    <h2>{{student.firstName}} {{student.lastName}} - Student: {{student.studentNum}}</h2>
    <hr />

    <form method="post" action="/student/update">
      <input type="hidden" name="studentNum" value="{{student.studentNum}}" />

      <fieldset>
        <legend>Personal Information</legend>
        <div class="row">
          <div class="col-md-6">
            <div class="form-group">
              <label>First Name:</label>
              <input class="form-control" name="firstName" type="text" value="{{student.firstName}}" />
            </div>
          </div>
          <div class="col-md-6">
            <div class="form-group">
              <label>Last Name:</label>
              <input class="form-control" name="lastName" type="text" value="{{student.lastName}}" />
            </div>
          </div>
        </div>
      </fieldset>

      <hr />

      <input type="submit" class="btn btn-primary float-right" value="Update Student" /><br /><br /><br />
    </form>
  </main>
```

```
</div>
```

- Once this is done, update your GET "/student/:studentNum" route according to the following specification
    - Use **res.render("student", { student: data });** inside the **.then()** callback (instead of res.json)
- Test the server **(/student/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
    - **Email**: type: "email", name: "email"
    - **Address (Street):** type: "text", name: "addressStreet"
    - **Address (City):** type: "text", name: "addressCity"
    - **Address (Province):** type: "text", name: "addressProvince"
    - **TA:** type: "checkbox", name: "TA", **(HINT**: **use the #if helper - {{#if data.TA}} … {{/if}} to see if the checkbox should be checked or not)**
    - **Status:** type: "radio" name: "status", values: "Full Time" or "Part Time" **(HINT**, **use the #equal helper - {{#equal data.status "Full Time" }} … {{/equal}} to see if Full Time or Part Time is checked)**
    - **Course** type: "select", name: "course", values: 1 - 11 inclusive **(HINT**, **use the #equal helper - {{#equal data.course "1" }} … {{/equal}} to determine which <option> should be selected)**
- 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: **/student/update** in our server.js file:

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

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

Step 2: Updating the collegeData.js module

- Add the new method: **updateStudent(studentData)** that **returns a promise**. This method will:
    - Search through the "students" array for an student with an studentNum that matches the JavaScript object (parameter studentData).
    - When the matching student is found, overwrite it with the new student passed in to the function (parameter studentData)
        - NOTE: Do not forget to correctly handle the "TA" checkbox data
    - Once this has completed successfully, invoke the **resolve()** method without any data.
- Now that we have a new **updateStudent()** method, we can invoke this function from our newly created **app.post("/student/update", (req, res) => { … });** route. Simply invoke the **updateStudent**() method with the **req.body** as the parameter. Once the promise is resolved use the **then()** callback to execute the **res.redirect("/students");** code.
- Test your server in the browser by updating Student 21 (Rozalie Dron). Once you have clicked "Update Student" and are redirected back to the student list, Student 21 should show your changes!

Part 5: 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
- Push your code to Heroku using the command **git push heroku master**
- **NOTE:** If you have decided to create a new Heroku application for this assignment, you can follow the "Heroku Guide" on the course website: https://web700.ca/getting-started-with-heroku

Testing: Sample Solution

To see a completed version of this app running, visit: https://web700-a5-example.herokuapp.com

Assignment Submission:

- Add the following declaration at the top of your **server.js** file:

```
/*********************************************************************************
* WEB700 – Assignment 05
* 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 assignment folder and submit the .zip file to My.Seneca under **Assignments** -> **Assignment 5**
- Submit your project file (ZIP) online using My.Seneca along with GitHub link, screenshots and small video recording with appropriate name for all files. Also submit heroku link if required. Please submit each file as separate submission.
-

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.