WEB322 Assignment 6

Assessment Weight:

9% of your final course Grade

Objective:

Update the given skeleton code to work with a Postgres data source on the server and add some new routes and functionality.

Specification:

NOTE: Please note: the "home", "about" and "htmlDemo" html files and the theme.css file will not be included in the skeleton given. You are add necessary CSS (specially navigation) and write code to make this assignment complete. Install necessary modules before you start.

Getting Started:

Before we get started, we must add a new Postgres instance on our WEB322 A6 app:

- Follow week 7 notes from web322.ca
- Record all of the credentials (ie: Host, Database, User, Password) - we will be using them to connect to the database:

Getting Started - Cleaning the solution

- To begin: open your Assignment 6 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 **collegeData.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.getAllStudents, module.exports.getStudentByNum, 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:
    - sequelize
    - pg

- pg-hstore

- At the top of your **collegeData.js** module, add the lines:

  - **const** Sequelize = require('sequelize');

    ```
    var sequelize = new Sequelize('database', 'user', 'password', {
      host: 'host',
      dialect: 'postgres',
      port: 5432,
      dialectOptions: {
        ssl: { rejectUnauthorized: false }
      },
      query:{ raw: true }
    });
    ```

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

Creating Data Models

- Inside your **collegeData.js** module (before your module.exports functions), define the following 2 data models and their relationship (**HINT**: See "Models (Tables) Introduction" in the Notes for examples)

- Student

Column Name	Sequelize DataType
studentNum	Sequelize.INTEGER primaryKey autoIncrement
firstName	Sequelize.STRING
lastName	Sequelize.STRING
email	Sequelize.STRING
addressStreet	Sequelize.STRING
addressCity	Sequelize.STRING
addressProvince	Sequelize.STRING
TA	Sequelize.BOOLEAN
status	Sequelize.STRING

- Course

Column Name	Sequelize DataType
courseId	Sequelize.INTEGER primaryKey autoIncrement
courseCode	Sequelize.STRING

courseDescription	Sequelize.STRING

- **hasMany** Relationship

  Since a course can have many students, we must define a relationship between Students and Courses, specifically:

  Course.hasMany(Student, {foreignKey: 'course'});

  This will ensure that our Student model gets a "course" column that will act as a foreign key to the Course model. When a Course is deleted, any associated Students will have a "null" value set to their "course" foreign key.

Update Existing collegeData.js functions

Now that we have Sequelize set up properly, and our "Student" and "Course" models defined, we can use all of the Sequelize operations, discussed in the Notes to update our collegeData.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 Student and Course models are represented in the database as tables.
- If the **sync()** 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 for the promise and pass an appropriate message, ie: reject("unable to sync the database").

getAllStudents()

- This function will invoke the Student.findAll() function
- If the **Student.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".

getStudentsByCourse(***course***)

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

getStudentByNum(***num***)

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

getCourses()

- This function will invoke the Course.findAll() function
- If the **Course.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".

getCourseById(***id***)

- Similar to the getStudentsByNum(***num***) function, this function will invoke the Course.findAll() function (instead of Student.findAll()) and filter the results by "id" (using the value passed to the function - ie: 1 or 2 or 3 … etc
- If the **Course.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".

addStudent(***studentData***)

- Before we can work with ***studentData*** correctly, we must once again make sure the TA property is set properly. Recall: to ensure that this value is set correctly, before you start working with the studentData object, add the line:
    - **studentData.TA = (studentData.TA) ? 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 province value for their address (causing studentData.addressProvince to be ""), this needs to be set instead to null (ie: studentData.addressProvince = 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 TA is explicitly set (true or false), and all of the remaining "" are replaced with null, we can invoke the Student.create() function
- If the **Student.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 student".

updateStudent(***studentData***)

- Like addStudent(studentData) we must ensure that the ***TA*** value is explicitly set to true/false and any blank values in ***studentData*** are set to null (follow the same procedure)
- Now that all of the "" are replaced with null, we can invoke the Student.update() function and filter the operation by "studentNum" (ie studentData.studentNum)
- If the **Student.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 student".

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 students in our "Students" view, and no courses within our "Courses" view. This is to be expected (since there is nothing in the database), however we are not seeing any error messages (just empty tables). To solve this, we must update our server.js file:

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

      we must place a condition there first so that it will only render "students" if data.length > 0. Otherwise, render the page with an error message,
        - ie: res.render("students",{ message: "no results" });
    - If we test the server now, we should see our "no results" message in the /students route
    - **NOTE**: We must still show messages if the promise(s) are rejected, as before
- /courses route
    - Using the same logic as above (for the /students route) update the /courses route as well
    - If we test the server now, we should see our "no results" message in the /courses 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 Student" link into its related page (ie: "add Student" will be moved out of the Navbar and into the "students" view.

- "add Student"
    - Remove this link ({{#navLink}} … {{/navLink}}) from the "navbar-nav" element inside the main.hbs file for "Add Student"

    - Inside the "students.hbs" view (Inside the <h2>Students</h2> element), add the below code to create a "button" that links to the "/students/add" route:
        - Add New Student

- "add Course"
    - You will notice that currently, we have no way of adding a new course. However, while we're adding our "add" buttons, it makes sense to create an "add Course" button as well (we'll code the route and collegeData.js function later in this assignment).
    - Inside the "courses.hbs" view (Inside the <h2>Courses</h2> element), add the below code to create a "button" that links to the "/courses/add" route:
        - Add New Course

Adding new collegeData.js functions

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

addCourse(***courseData***)

- Like addStudent(studentData) function we must ensure that any blank values in ***courseData*** are set to null (follow the same procedure)
- Now that all of the "" are replaced with null, we can invoke the Course.create() function
- If the **Course.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 course"

updateCourse(***courseData***)

- Like addCourse(courseData) function we must ensure that any blank values in ***courseData*** are set to null (follow the same procedure)
- Now that all of the "" are replaced with null, we can invoke the Course.update() function and filter the operation by "courseId" (ie courseData.courseId)
- If the **Course.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 course".

deleteCourseById(***id***)

- The purpose of this method is simply to "**delete**" courses using the Course.destroy() for a specific course by "id" (courseId). Ensure that this function returns a **promise** and only "resolves" if the Course was deleted ("destroyed"). "Reject" the promise if the "destroy" method encountered an error (was rejected).

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

Now that we have our collegeData.js module up to date to deal with course 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).

/courses/add

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

/courses/add

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

/course/update

- This **POST** route is very similar to your current "/student/update" route - only instead of calling the updatestudent()function, you will instead call your newly created updateCourse() function with the POST data.
- Instead of redirecting to /students when the promise has resolved (using .then()), we will instead redirect to /courses

/course/:id

- Since we already have this route created, there's very little that we need to change. However, we do need to check if the data returned by "getCourseById" exists or not before we try to render it (otherwise, we'll get a blank form). To achieve this, first check if the data is undefined (ie, no error occurred, but no results were returned either). If it is, send a 404 error back to the client using: **res.status(404).send("Course Not Found").** Otherwise, render the "course" view as before.

/course/delete/:id

- This **GET** route will invoke your newly created **deleteCourseById(id)** collegeData.js function. If the function resolved successfully, redirect the user to the **"/courses"** view. If the operation encountered an error, return a **status code of 500** and the plain text: **"Unable to Remove Course / Course not found)"**

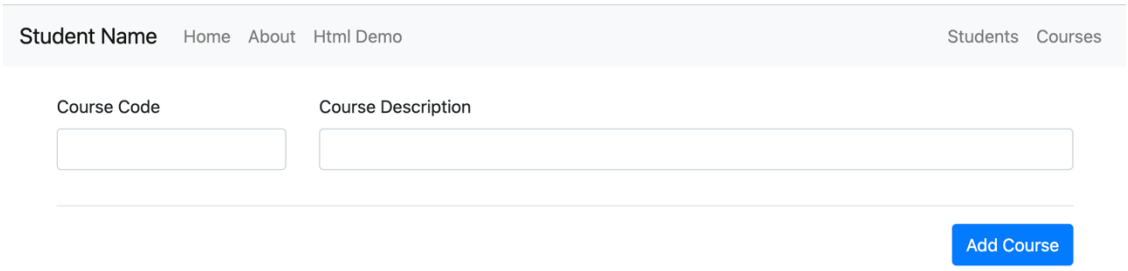
Updating Views to Add / Update & Delete Courses

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

addCourse.hbs

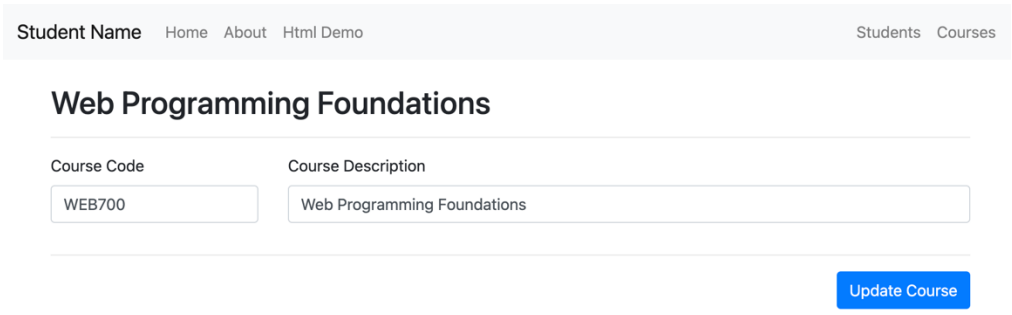
- Fundamentally, this view is nearly identical to the **addStudent.hbs** view, however there are a few key changes:
    - The form must submit to "/courses/add"
    - There must be only two input fields
        - type: "text", name: "courseCode", label: "Course Code")

        - type: "text", name: "courseDescription", label: "Course Description")
    - The submit button must read "Add Course"
- When complete, your view should appear as:



course.hbs

- We should already have a course.hbs file. However, we must make a few changes to allow users to actually modify the course code & description
    - The form method "POST" must be added, and the form must submit to "/course/update"
    - The "readonly" property from the "courseCode" & "courseDescription" fields must be removed
    - A "submit" button must be added at the end of the form that reads "Update Course" (**HINT:** You can use same button from the "student.hbs" view, just change the text to "Update Course" instead of "Update Student")
    - The form must include a hidden field with the properties: name="courseId" and value="{{course.courseId}}" in order to keep track of exactly which course the user is editing
    - When complete, your view should appear as the following (if we're currently looking at a newly created "WEB700", for example):



courses.hbs

- Lastly, to enable users to access all of this new functionality, we need to make one important change to our current courses.hbs file:
    - **Add** a "remove" link for every course within in a new column of the table (at the end) - Note: The header for the column should not contain any text (ie: <hr></hr>). The links in every row should be styled as a button (ie: class="btn btn-danger") with the text "remove" and simply link to the newly created GET route "course/delete/*courseId*" where *courseId* is the courseId of the course in the current row. Once

this button is clicked, the course will be deleted and the user will be redirected back to the "/courses" list. (Hint: See the sample solution if you need help with the table formatting)

Now, users can click on the course code if they wish to edit an existing course, the course description if they wish to filter the student list by course, or "delete" if they wish to remove the course!

Updating the "Course" List in the Student Views

Now that users can add new Courses, it makes sense that all of the Courses available to a student (either when adding a new student or editing an existing one), should consist of all the current courses in the database (instead of just 1…11). To support this new functionality, we must make a few key changes to our routes and views:

"/students/add" GET route

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

"addStudent.hbs" view

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

```
{{#if courses}}
    <select class="form-control" name="course" id="course">
        <option value="">Select Course</option>
        {{#each courses}}
            <option value="{{courseId}}">{{courseCode}}</option>
        {{/each}}
    </select>
{{else}}
    <div>No Courses</div>
{{/if}}
```

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

"/student/:studentNum" route

- If we want to do the same thing for existing students, the task is more complicated: In addition to sending the **current Student** to the "student" view, we must also send all of the **Courses**. This requires two separate calls to our collegeData module ("data" in the below code) that return data that needs to be sent to the view. Not only

that, but we must ensure that the right course is selected for the student 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("/student/:studentNum", (req, res) => {

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

  data.getStudentByNum(req.params.studentNum).then((data) => {
    if (data) {
      viewData.student = data; //store student data in the "viewData" object as "student"
    } else {
      viewData.student = null; // set student to null if none were returned
    }
  }).catch(() => {
    viewData.student = null; // set student to null if there was an error
  }).then(data.getCourses)
  .then((data) => {
    viewData.courses = data; // store course data in the "viewData" object as "courses"

    // loop through viewData.courses and once we have found the courseId that matches
    // the student's "course" value, add a "selected" property to the matching
    // viewData.courses object

    for (let i = 0; i < viewData.courses.length; i++) {
      if (viewData.courses[i].courseId == viewData.student.course) {
        viewData.courses[i].selected = true;
      }
    }

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

"student.hbs" view

- Now that we have all of the data for the student inside "viewData.student" (instead of just "student"), we must update every handlebars reference to data, from ***student.propertyName*** to ***viewData.student.propertyName***. For example: **{{student.firstName}}** would become: **{{viewData.student.firstName}}**

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

```
{{#if viewData.courses}}
  <select class="form-control" name="course" id="course">
```

```
    <option value="">Select Course</option>
{{#each viewData.courses}}
    <option value="{{courseId}}" {{#if selected}} selected {{/if}} >{{courseCode}} </option>
  {{/each}}
  </select>
{{else}}
  <div>No Courses</div>
{{/if}}
```

Updating server.js, collegeData.js & students.hbs to Delete Students

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

- Creating a new function (ie: **deleteStudentByNum(studentNum)**) in collegeData.js to "**delete**" students using the Student.destroy() for a specific student. Ensure that this function returns a **promise** and only "resolves" if the Student was deleted ("destroyed"). "Reject" the promise if the "destroy" method encountered an error (was rejected).
- Create a new GET route (ie: "**/student/delete/:studentNum**") that will invoke your newly created **deleteStudentByNum(studentNum)** collegeData method. If the function resolved successfully, redirect the user to the **"/students"** view. If the operation encountered an error, return a **status code of 500** and the plain text: **"Unable to Remove Student / Student not found)"**
- Lastly, update the **students.hbs** view to include a "remove" link for every student 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 "**students/delete/*studentNum***" where ***studentNum*** is the student number of the student in the current row. Once this button is clicked, the student will be deleted and the user will be redirected back to the **"/students**" list.

Managing State

Follow week 10 notes to setup a login route for this assignment application. Use “sampleuser” for username and “samplepassword” for password (Do not use anything else). You must ensure that the user is logged in to protect against unauthorized access to any of the routes. Client sessions need to be maintained for this. You must have the login route appearing in the navigation (header) area. Any JS and html code writing is up to your discretion for this section. But keep the design simple (login/logout etc.).

Pushing to Cyclic

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

- Ensure that you have checked in your latest code using **git** (from within Visual Studio Code)
- Open the integrated terminal in Visual Studio Code
- **NOTE:** If you have decided to create a new Cyclic application for this assignment, you can follow the "Cyclic Guide" on the course website.

Assignment Submission

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

```
/*********************************************************************************
* WEB322 – Assignment 06
* 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 web sites) or
* distributed to other students.
*
* Name: _____ Student ID: _____ Date: _____
*
* Online (Cyclic) Link: _____
*
********************************************************************************/
```

- Compress (.zip) your assignment folder (optionally omitting node_modules, to make the upload faster) and submit the .zip file to My.Seneca under **Assignments** -> **Assignment 6**
- **Type your working Cyclic link in the comment box when you upload in blackboard. Failure to do so will result in zero mark for this assignment.**

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 (11:30PM) 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.