WEB322 Assignment 3

Submission Deadline:

Friday, December 5th @ 11:59 PM

Assessment Weight:

10% of your final course Grade

Objective:

Build upon Assignment 2 by refactoring our code to use a Postgres database to manage our Project Data, as well as enable the creation, modification and deletion of Projects in the collection.

If you require a *clean version* of Assignment 2 to begin this assignment, please email your professor.

NOTE: Please reference the sample: https://wptf-a3-fall-2025.vercel.app (log in with user: "admin" password: "admin") when creating your solution. Once again, the UI does not have to match exactly, but this will help you determine which elements / syntax should be on each page. **You may copy** any HTML / CSS code from here if it helps with your solution.

Additionally, since this sample is shared with all students in the class, please **do not add** any content to the Project collection that may be considered harmful or disrespectful to other students.

Part A: Postgres Integration

Step 1: Connecting to the DB & Adding Existing Projects

Since the major focus of this assignment will be refactoring our code to use a Postgres database, let's begin with this. Follow the course notes PostgreSQL (Postgres) to set up a database on Vercel with Neon and record the following information.

- PGHOST
- PGUSER
- PGDATABASE
- PGPASSWORD

Now, with your assignment folder open, add the file ".env" in the root of your solution and add the text that you copied from Neon, ie:

PGHOST=**host**
PGUSER=**user**
PGDATABASE=**database**
PGPASSWORD=**password**

Where: **host**, **user**, **database** and **password** are the specific values from your Neon database

To actually connect to the database and use the .env file, we must install the Sequelize, pg, pg-hstore and dotenv modules from NPM: **npm install sequelize pg pg-hstore dotenv**

Finally, before we write our Sequelize code, open the file: /modules/projects.js and add the "dotenv" module at the top using the code:

require('dotenv').config();

This will allow us to access the PGUSER, PGDATABASE, etc. values from the ".env" file using the "process.env" syntax, ie: **process.env.PGUSER**, **process.env.PGDATABASE** etc.

Beneath this line, add the code to include the "sequelize" module (and ensure it works on Vercel):

require('pg');
const Sequelize = require('sequelize');

and create the "sequelize" object only using **let sequelize = new Sequelize(…);** - see: "Getting Started" in the Relational Database (Postgres) Notes. Be sure to include all of the correct information using **process.env** for "database", "user", "password" and "host".

With our newly created "sequelize" object, we can create the two "models" required for our Assignment according to the below specification (Column Name / Sequelize Data Type):

NOTE: We also wish to disable the **createdAt** and **updatedAt** fields – see: Models (Tables) Introduction

- **Sector**: const Sector = sequelize.define('Sector', { ... });

Column Name	Sequelize DataType
id	Sequelize.INTEGER primaryKey (true) autoincrement (true)
sector_name	Sequelize.STRING

- **Project:** const Project = sequelize.define(Project, { ... });

Column Name	Sequelize DataType
id	Sequelize.INTEGER primaryKey (true) autoincrement(true)
title	Sequelize.STRING
feature_img_url	Sequelize.STRING
summary_short	Sequelize.TEXT
intro_short	Sequelize.TEXT
impact	Sequelize.TEXT
original_source_url	Sequelize.STRING

Now that the models are defined, we must create an association between the two:

Project.belongsTo(Sector, {foreignKey: sector_id'});

Adding Existing Projects using "BulkInsert"

With our models correctly defined, we have everything that we need to start working with the database. To ensure that our existing data is inserted into our new "Sectors" and "Projects" tables, copy the code from here:

https://pat-crawford-sdds.netlify.app/shared/fall-2025/web322/A3/bulkInsert.txt

and insert it at the bottom of the **/modules/projects.js** file (beneath all module.exports)

(**NOTE:** this code snippet assumes that you have the below code from Assignment 3 still in place):

```
const projectData = require("../data/projectData");
const sectorData = require("../data/sectorData");
```

With the code snippet from the above URL in place, open the integrated terminal and execute the command to run it:

node modules/projects.js

This should show a big wall of text in the console, followed by "data inserted successfully"!

Step 2: Refactoring Existing Code to use Sequelize

Now that all of our projects exist on the database, we can refactor our existing code in the **projects.js** module to retrieve them. This can be done by following the below steps:

1. **Delete** the above code snippet to "bulkInsert" (we no longer need it, now that our data is available in the database)
2. **Delete** the code to read the JSON files / initialize an empty "projects" array:

   ```
   const projectData = require("../data/projectData");
   const sectorData = require("../data/sectorData");
   let projects = [];
   ```

3. **Change** your code in "initialize" to instead invoke **sequelize.sync()**. If sequelize.sync() resolves successfully, then we can resolve the returned Promise, otherwise reject the returned Promise with the error.
4. **Change** your code in the "getAllProjects()" function to instead use the "Project" model (defined above) to resolve the returned Promise with all returned projects (see: Operations (CRUD) Reference).

   NOTE: Do not forget the option **include: [Sector]** to include Sector data when invoking "findAll".

5. **Change** your code in the "getProjectById(projectId)" function to instead use the "Project" model (defined above) to resolve the returned Promise with a single project whose id value matches the "projectId" parameter. As before, if no project was found, reject the Promise with an error, ie: "Unable to find requested project"

NOTE: Do not forget the option **include: [Sector]** to include Sector data when invoking "findAll". Also, remember to resolve with the **first** element of the returned array ([0]) to return a single object, as "findAll" always returns an array

6. **Change** your code in the "getProjectsBySector(sector)" function to instead use the "Project" model (defined above) to resolve the returned Promise with all the returned projects whose "Sector.sector_name" property contains the string in the "sector" parameter. This will involve a more complicated "where" clause, ie:

```
Project.findAll({include: [Sector], where: {
  '$Sector.sector_name$': {
    [Sequelize.Op.iLike]: `%${sector}%`
  }
}});
```

   As before, if no projects were found, reject the Promise with an error, ie: "Unable to find requested projects"

   NOTE: We have once again included the option **include: [Sector]** to include Sector Data.

7. Look through your .ejs files for "project.sector" and instead replace it with: "project.Sector.sector_name". This is because when "including" the Sector model in our above functions, we have added a full "Sector" object to the result objects (project / projects).

8. Test your code by running the usual **node server.js** – it should work exactly as before!

Step 3: Adding New Projects

Since we are now using a database to manage our data, instead of JSON file(s), the next logical step is to enable users to Create / Update and Delete project data. To begin, we will first create the logic / UI for creating projects and will focus on editing and deleting in the following steps.

Creating the Form

To begin, we should create a simple UI with a form according to the following specification

- This should be in a new file under "views", ie "/views/addProject.ejs"
- It should have some kind of title / header / hero, etc. text to match the other views, ie: "Add Project"
- It should render the "navbar" partial, ie:
  <%- include('partials/navbar') %>
- It must have the following form controls and submit using "POST" to "/solutions/addProject" (to be created later)

NOTE: The HTML in the sample code may be used here to help render the form (if you wish)

- **title**
  - input type="text"
  - required
- **feature_img_url**
  - input type="url"
  - required
- **sector_id**
  - select
  - required
  - each option must be a "sector", ie:

```
<option value="1">Land Sinks</option>
<option value="2">Industry</option>
<option value="3">Transportation</option>
<option value="4">Electricity</option>
<option value="5">Food, Agriculture, and Land Use</option>
```

- **intro_short**
  - textarea
  - required
- **summary_short**
  - textarea
  - required
- **impact**
  - textarea
  - required
- **original_source_url**
  - input type="url"
  - required
- **Submit Button**

NOTE: Do not forget to run the command **npm run tw:build** (if it's not already running) after creating the form, as new CSS was likely used.

Adding a New View: "500.ejs"

Since it's possible that we may encounter database errors, we should have some kind of "500" error message to show the user instead of rendering a regular view. To get started, make a copy of your "404.ejs" file and update it to show the text "500" as well as any other cosmetic updates you would like to use.

Small Navbar Update

At the moment, there is no way for a user to view all projects without first going to the default "/" route and clicking "Get Started Now" (other than manually entering the url). To fix this, we will add another "View All" item to the navbar (beneath the other sectors). If you wish to have a spacer like the sample, you can use the following code: (**Note**, the colour of the line can be changed by modifying / removing the "border-gray-500" class):

```
<li class="border-t border-gray-500 mt-2 pt-2"><a href="/solutions/projects">View All</a></li>
```

Creating the routes in server.js

To correctly serve the "/solutions/addProject" view and process the form, two routes are required in your server.js code (below).

Additionally, since our application will be using urlencoded form data, the "**express.urlencoded({extended:true})**" middleware should be added

- GET /solutions/addProject

  This route must render the "addProject" view.

- POST /solutions/addProject

  This route must make a request to a Promise-based "addProject(projectData)" function (to be added later in the projects.js module), and provide the data in **req.body** as the "projectData" parameter.

  Once the Promise has resolved successfully, redirect the user to the "/solutions/projects" route.

  If an error was encountered, instead render the new "500" view with an appropriate message, ie:

  res.render("500", { message: `I'm sorry, but we have encountered the following error: ${err}` });

  NOTE: we do not explicitly set the "500" status code here, as it causes unexpected behaviour on Vercel.

Adding new functionality to projects.js module

In our new routes, we made some assumptions about upcoming functionality to be added in the "projects.js" module, specifically: "addProject(projectData)" and "getAllSectors()". To complete the functionality to add new projects, let's create these now:

NOTE: do not forget to use / add to "module.exports" to make these functions available to server.js

- addProject(projectData)

  This function must return a Promise that resolves once a project has been created, or rejects if there was an error.

  It uses the "Project" model to create a new Project with the data from the "projectData" parameter. Once this function has resolved successfully, resolve the Promise returned by the addProject(projectData) function

without any data.

However, if the function did not resolve successfully, reject the Promise returned by the addProject(projetData) function with the message from the *first* error, ie: **err.errors[0].message** (this will provide a more human-readable error message)

Adding an Add button

Finally, add an "Add Project" link (rendered using the tailwind button classes, ie: "btn btn-success", etc) in the **projects.ejs** template that links to: "/solutions/addProject/" somewhere on your "Projects" view. The sample solution has this in the "hero" section.

Test the Server

We should now be able to add new projects to our collection.

Step 4: Editing Existing Projects

In addition to allowing users to **add** projects, we should also let them **edit** existing projects. Let's try to follow the same development methodology used when creating the functionality for adding new projects. This means starting with the view:

- This should be in a new file under "views", ie "/views/editProject.ejs"
- It should have some kind of title / header / hero, etc. text to match the other views, ie: "Edit Project" followed by the **project title**, ie **<%= project.title %>**
- It should render the "navbar" partial, ie: **<%- include('partials/navbar') %>**
- It must have the exact form controls as the "addProject" view, however the values must be populated with data from a "project" object, passed to the view. For each most of the controls, this will simply mean setting a "value" attribute, ie:

  value="<%= project.title %>" or **<textarea><%= project.intro_short %></textarea>,** etc.

  However, things get more complicated when we wish to correctly set the "selected" attribute of the "sector_id" select control. For example:

  <option value="1" <%= (project.sector_id == 1) ? "selected" : "" %>>Land Sinks</option>

- Ensure that a hidden "id" field is added to keep track of the current Project being edited
- Finally, we should change the "action" attribute on the <form> control to submit the form to **"/solutions/editProject"** as well as change the submit button text to something like "Update Project"

Creating the routes in server.js

To correctly serve the "/solutions/editProject" view and process the form, two routes are required in your server.js code (below).

- GET "/solutions/editProject/:id"

  This route must make a request to the Promise-based "getProjectById(projectId)" function with the value from the **id** route parameter as "projectId" in order to retrieve the correct project.

  Once the Promise has resolved, the "edit" view must be rendered with the data, ie:

  res.render("editProject", {project: projectData });

  However, if there was a problem obtaining the project (ie: the Promise was rejected), instead render the "404" view with an appropriate message, ie:

  res.status(404).render("404", { message: err });

- POST "/solutions/editProject"

  This route must make a request to a Promise-based "editProject(id, projectData)" function (to be added later in the projects.js module), and provide the data in **req.body.id** as the "id" parameter and **req.body** as the "projectData" parameter

  Once the Promise has resolved successfully, redirect the user to the "/solutions/projects" route.

  If an error was encountered, instead render the "500" view with an appropriate message, ie:

  res.render("500", { message: `I'm sorry, but we have encountered the following error: ${err}` });

  NOTE: we do not explicitly set the "500" status code here, as it causes unexpected behaviour on Vercel.

Adding new functionality to projects.js module

In our new routes, we made some assumptions about upcoming functionality to be added in the "projects.js" module, specifically: "editProject(id, projectData)". To complete the functionality to edit projects, let's create this now:

NOTE: do not forget to use / add to "module.exports" to make the function available to server.js

- editProject(id, projectData)

  This function must return a Promise that resolves once a project has been updated, or rejects if there was an error.

  It uses the "Project" model to update an existing project that has a "id" property that matches the "id" parameter to the function, with the data from the "projextData" parameter. Once this function has resolved successfully, resolve the Promise returned by the editProject(id, projectData) function without any data.

  However, if the function did not resolve successfully, reject the Promise returned by the editProject(id,projectData) function with the message from the *first* error, ie: **err.errors[0].message** (this will provide a more human-readable error message)

Adding an Edit button

At the moment, we should be able to edit any of our projects by going directly to the route in the browser, ie: "/solutions/editProject/1" However, it makes more sense from a usability perspective to allow users the ability to navigate to this route using the UI.

To achieve this, add an "edit" link (rendered using the tailwind button classes, ie: "btn btn-success", etc) in the **project.ejs** template that links to: "/solutions/editProject/**id**" where **id** is the "id" value of the current project, ie: **<%= project.id %>**

Test the Server

We should now be able to edit projects!

Step 5: Deleting Existing Projects

Next, we should enable users to remove (delete) an existing project from the database. To achieve this, we must add an additional function on our **projects.js** module:

NOTE: do not forget to use / add to "module.exports" to make the function available to server.js

- deleteProject(id)

  This function must return a Promise that resolves once a project has been deleted, or rejects if there was an error.

  It uses the "Project" model to delete an existing project that has an "id" property that matches the "id" parameter to the function. Once this function has resolved successfully, resolve the Promise returned by the deleteProject(id) function without any data.

  However, if the function did not resolve successfully, reject the Promise returned by the deleteProject(id) function with the message from the *first* error, ie: **err.errors[0].message** (this will provide a more human-readable error message)

With this function in place, we can now write a new route in server.js :

- GET "/solutions/deleteProject/:id"

  This route must make a request to the Promise-based "deleteProject(id)" function with the value from the **id** route parameter as "id" in order to delete the correct project.

  Once the Promise has resolved successfully, redirect the user to the "/solutions/projects" route.

  If an error was encountered, instead render the "500" view with an appropriate message, ie:

  res.render("500", { message: `I'm sorry, but we have encountered the following error: ${err}` });

  NOTE: we do not explicitly set the "500" status code here, as it causes unexpected behaviour on Vercel.

Finally, let's add a button in our UI to enable this functionality by linking to the above route for the correct project. One place where it makes sense is in our "**editProject.ejs**" view. Here, we give the user the choice to either update the project or delete it.

To achieve this, add a "Delete Project" link (rendered using the tailwind button classes, ie: "btn btn-error", etc) that links to: "/solutions/deleteProject/**id**" where **id** is the "id" value of the current project, ie: **<%= project.id %>**

Test the Server

We should now be able to remove projects!

Part B: Logging In and Securing Routes

Step 1: ClientSessions Configuration

The first thing we must to secure our routes is to download and "require" the "client-sessions" module using NPM and correctly configure our app to use the middleware:

1. Open the "Integrated Terminal" in Visual Studio Code and enter the command:
   npm install client-sessions
2. Be sure to "require" the new "client-sessions" module at the top of your **server.js file** as **clientSessions**.
3. Add a SESSIONSECRET value to your .env file, ie: **SESSIONSECRET=o6LjQ5EVNC28ZgK64hDELM18ScpFQr**
4. Ensure that we correctly use the client-sessions middleware with appropriate **cookieName, secret (using process.env.SESSIONSECRET, duration** and **activeDuration** properties (**HINT**: Refer to the "Middleware" notes under "Managing State Information")
5. Once this is complete, incorporate the following custom middleware function to ensure that all of your templates will have access to a "session" object (ie: <%= session.user?.userName %> for example) - we will need this to conditionally hide/show elements to the user depending on whether they're currently logged in.

   ```
   app.use((req, res, next) => {
     res.locals.session = req.session;
     next();
   });
   ```

6. Define a helper middleware function (ie: **ensureLogin** from the "Middleware" notes, in the "Practical Application" section) that checks if a user is logged in (we will use this in all of our post / category routes). If a user is not logged in, redirect the user to the "/login" route.
7. Update all routes that allow users to **add**, **edit**, or **delete** Projects (this should be 5 routes) to use your custom **ensureLogin** helper middleware.

Step 2: Logging in as Admin

For this assignment, we will have a single user that will log into the system, "admin". To begin, in your .env file, add the two values:

ADMINUSER=admin
ADMINPASSWORD=admin

Note: We will keep these values as "admin" for now so that the assignment can be evaluated without changing your code. However, once the assignment has been graded, please feel free to change these values.

Next, make sure your server.js uses "dotenv", ie: **require('dotenv').config();**

Adding The Login View

Next, we will create a "Login" View with the following form:

- This (new) view must consist of the "login form" which will allow the user to submit their credentials (using **POST**) to the **"/login"** POST route:

input type	Properties	Value
text	name: "userName" placeholder: "User Name" required	**userName** if it was rendered with the view. Refer to the "/login" POST route below for more information
password	name: "password" placeholder: "Password" required	
submit (button)	text / value: "Login"	

- Below the form, we must have a space available for error output: Show the daisyUI Error Alert:

```
<div role="alert" class="alert alert-error">
  <svg xmlns="http://www.w3.org/2000/svg" class="h-6 w-6 shrink-0 stroke-current" fill="none" viewBox="0 0 24 24">
    <path stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="M10 14l2-2m0 0l2-2m-2 2l-2-2m2 2l2 2m7-2a9 9 0 11-18 0 9 9 0 0118 0z" /></svg>
  <span><%= errorMessage %></span>
</div>
```

only if there is an errorMessage rendered with the view.

Adding New Routes:

With our app now capable of respecting client sessions and communicating with MongoDB to register/validate users, we need to create **routes** that enable the user to register for an account and login / logout of the system (above our 404 middleware function). Once this is complete, we will create the corresponding **views** (Step 6).

GET /login

- This "GET" route simply renders the "**login**" view with the data: { errorMessage: "", userName: "" }

POST /login

- This "POST" route will check the values of the "userName" and "password" fields and ensure that they both match the static ADMINUSER (process.env.ADMINUSER) and ADMINPASSWORD (process.env.ADMINPASSWORD) values
    - If the user and password both match, add the process.env.ADMIN user value to the session as "userName" and redirect the user to the "/solutions/projects" view, ie:

```
req.session.user = {
   userName: process.env.ADMINUSER
}

res.redirect('/solutions/projects);
```

    - If values don't match, **render** the **login** view with the following data: **{errorMessage: 'Invalid User Name or Password', userName: req.body.userName} - NOTE:** we are returning the user back to the page, so the user does not forget the **user value** that was used to attempt to log into the system

GET /logout

- This "GET" route will simply "reset" the session (**Hint**: refer to the "Route Updates (Logic)" section) and redirect the user to the "/" route, **ie: res.redirect('/');**

Step 3: Updating the UI

Currently, unauthorized users can still see the "Add New Project" and "Edit Project" buttons, even when they are not logged in. Additionally, there is no "Log Out" button available anywhere in the UI. To remedy this, we must:

- In the "Projects" view, **only** show the "Add New Project" button if there is a value for "session.user", ie:

```
<% if(session.user){ %>
    ...
<% } %>
```

- In the "Project" view, **only** show the "Edit Project" button if there is a value for "session.user"
- In the "Navbar" partial, add the following items:
    - "Log In" (which redirects to "/login") **only if** there is **not** a value for "session.user"
    - "Log Out (**user**)" - where **user** is the value for **session.user.userName** (which redirects to "/logout") **only if** there **is** a value for "session.user"

Part C: Deploying to Vercel

Finally, once you have tested your site locally and are happy with it, it's time to publish it online. First, ensure that your .env file is located in your .gitignore file. Once this is complete, follow the steps in the "Vercel Guide".

NOTE: Do not forget to add your environment variables (there should be 7 of them). To make this easier, Vercel has added an "import .env" option.

For more information on setting up environment variables on Vercel, see:
https://vercel.com/docs/projects/environment-variables

Assignment Submission:

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

```
/*********************************************************************************
* WEB322 – Assignment 03
*
* I declare that this assignment is my own work in accordance with Seneca's
* Academic Integrity Policy:
*
* https://www.senecacollege.ca/about/policies/academic-integrity-policy.html
*
* Name: _____ Student ID: _____ Date: _____
*
* Published URL: _____
*
********************************************************************************/
```

- Compress (.zip) your assignment folder and submit the .zip file to My.Seneca under **Assignments** -> **Assignment 3**

Important Note:

- **NO LATE SUBMISSIONS** for assignments. Late assignment submissions will not be accepted and will receive a **grade of zero (0)**.
- 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.