# WEB322 Assignment 5

## Submission Deadline:

Monday, March 24 – 11:59 PM

## Assessment Weight:

9% of your final course Grade

## Objective:

Build upon Assignment 4 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 4 to begin this assignment, please email your professor.

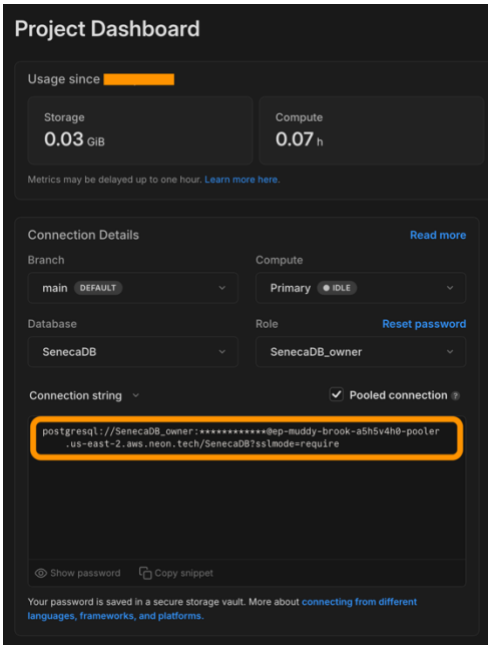
**NOTE**: Please refer the sample: https://web322-w25-a5.vercel.app/ when creating your solution. Once again, the UI does not have to match exactly, but this will help you determine which elements / syntax should be on each page. **You may copy** any HTML / CSS code from here if it helps with your solution.

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 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 https://neon.tech and record the following information.

- PG CONNECTION STRING



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 tech, ie:

PG_CONNECTION_STRING='**connection string**'

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 value of PG_CONNECTION_STRING from the ".env" file using the "process.env" syntax, ie: **process.env.PG_CONNECTION_STRING**

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 **const sequelize = new Sequelize( … );** - see: "Getting Started" in the Relational Database (Postgres) Notes. The code on the website uses an old way to connect to the DB. I have included the new snippet that uses the connection string instead. Be sure to include the correct information using **process.env** for PG_CONNECTION_STRING.

Get the new snippet from here: https://web322-w25.vercel.app/a5/sequelize.txt

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<br>primaryKey (true)<br>autoincrement (true) |
| sector_name | Sequelize.STRING |

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

| Column Name | Sequelize DataType |
|---|---|
| id | Sequelize.INTEGER<br>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://web322-w25.vercel.app/a5/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 above 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"!

## Part 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 Setor 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!

## Part 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 with a "page" value of "/solutions/addProject", ie:
  **<%- include('partials/navbar', {page: '/solutions/addProject }) %>**
- 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" (added to the view later), ie:

      ```
      <% sectors.forEach(sector=>{ %>
          <option value="<%= sector.id %>">
              <%= sector.sector_name %>
          </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** after creating the form, as new CSS was likely used.

## Updating the Navbar Partial

As you have noticed from the above steps, a small update is required to our navbar to support linking to the view & highlighting the navbar item. To achieve this, add the following navbar item where appropriate (ie: in the regular & responsive navbar HTML elements)

<li><a class="<%= (page == "/solutions/addProject") ? 'active' : '' %>" href="/solutions/addProject">Add Project</a></li>

## 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.

## 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 make a request to a Promise-based "getAllSectors()" function (to be added later in the projects.js module)

  Once the Promise has resolved with the sectors, the "addproject" view must be rendered with them, ie:
  **res.render("addProject", { sectors: sectorData })**;

- 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 "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)

- getAllSectors()

  This function must return a Promise that resolves with all the sectors in the database. This can be accomplished using the "Sector" model to return all of the sectors in the database

## Test the Server

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

## Part 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 with an empty "page" value "" (since this page will not be represented in the navbar), ie: **<%- include('partials/navbar', {page: '' }) %>**
- 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, ie:

  ```
  <% sectors.forEach(sector=>{ %>
    <option <%= (project.sector_id == sector.id) ? "selected" : "" %> value="<%= sector.id %>">
      <%= sector.sector_name %>
    </option>
  <% }) %>
  ```

  (once again this assumes that a "sectors" collection will be added to the view along with a "project" object later)
- 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.

It also must make a request to the Promise-based "getAllSectors() function in order to retrieve an array of "sector" data

Once the Promises have resolved with the sector data and the project data the "edit" view must be rendered with them, ie:

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

However, if there was a problem obtaining the project or collection of sectors (ie: the Promises were 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 "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!

## Part 5: Deleting Existing Projects

The final piece of logic that we will implement in this assignment is to 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 "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 6: Updating your Deployment

Finally, once you have tested your site locally and are happy with it, you're ready to update your deployed site by pushing your latest changes to GitHub.  However, before you do that, you should add **.env** to your **.gitignore** file to prevent your environment variables from being included:

**File: .gitignore**

node_modules
.env

Additionally, you should add those environment variable values to your app on Vercel by logging in and proceeding to the dashboard for your app and navigating to the "Settings" tab.  From there you can add or remove environment variables in the “Environment Variables” Section

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

Once this is complete, "push" your code to GitHub to update your deployment

## Assignment Submission:

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

  /**********************************************************************************
  * WEB322 – Assignment 05
  *
  * I declare that this assignment is my own work in accordance with Seneca's
  * Academic Integrity Policy:
  *
  * https://www.senecapolytechnic.ca/about/policies/academic-integrity-policy.html
  *
  * Name: _____ Student ID: _____ Date: _____
  *
  **********************************************************************************/

- Compress (.zip) your web322-app (after removing node_modules folder).
- Create a screen recording (**maximum 15 minutes**) that starts by showing your app in action (on Vercel URL). After the demo, explain the code, focusing on the flow and logic—avoid simply reading it line by line. Throughout the recording, narrate what you're doing in real time.
- **IMPORTANT**: The app demo in screen recording must be done on the hosted website (Vercel)
- **Submit the following:**
  - **Project zip file (without node_modules folder)**
  - **Published URL (host the project on Vercel and share the public link in submission comments)**
  - **Screen Recording URL (upload the video to YouTube, set it as "unlisted", and share the link in submission comments)**

  **Note:**
  - **All items listed above must be submitted for your submission to be considered complete/valid.**
  - **Incomplete/Invalid submissions will not be accepted and will receive a grade of zero (0).**

- Here is a video tutorial on how to upload videos to YouTube and set them as "unlisted": https://www.youtube.com/watch?v=jaftEW9WI3U
- Here is a video tutorial on how to host your web app on Vercel and get the public/hosted URL: https://www.youtube.com/watch?v=Rko7rMe4l60
- Here is a video tutorial on how to create multiple branches in Git and use them to create deployments on Vercel: https://www.youtube.com/watch?v=NyqzCuW7LMc

## 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.