WEB322 Assignment 2

Submission Deadline:

Friday, November 7th, 2025 @ 11:59 PM

Assessment Weight:

9% of your final course Grade

Objective:

Build upon Assignment 1 by adding a custom landing page with links to various projects, as well as an "about" page and custom 404 error page. Additionally, we will be updating our server.js file to support more dynamic routes, views, status codes and static content (css). Finally, we will publish the solution using Vercel.

If you require a *clean version* of Assignment 1 to begin this assignment, please email your professor.

NOTE: Please reference the sample: https://wptf-a2-fall-2025.vercel.app when creating your solution. The UI does not have to match exactly, but this will help you determine which elements / syntax should be on each page.

Part 1: Installing / Configuring Tailwind CSS

For this assignment, we will be adding multiple pages, including a landing page with links to some of your Projects. To make these appealing to the end user, we will be leveraging our knowledge of Tailwind CSS. With your Assignment 1 folder open in Visual Studio Code, follow the follow the steps identified in Tailwind CSS & daisyUI to set up Tailwind CSS, ie:

- Installing "tailwindcss" and "@tailwindcss/cli" as "devDependencies"
- Installing "@tailwindcss/typography" and "daisyui"
- Creating a "tailwind.css" file in /public/css
- Editing tailwind.css to include the required @import statement for "tailwindcss", as well as the @plugin statements for "@tailwindcss/typography" and "daisyui". Also, you may add a "theme" - the sample uses "dim", but you're free to use whatever you like.
- Adding a "tw:build" script to your package.json
- and finally building a main.css file.

Part 2: Installing / Configuring EJS

This assignment will be using .ejs files instead of .html files, since the content of *some* of the views will be dynamic, instead of static. If you are beginning this assignment before EJS has been formally discussed in class, please note that when creating static .ejs files (see: "Part 3"), the syntax is identical to regular HTML.

In order to use .ejs however, we need to:

- Using npm to "install" the ejs package: **npm install ejs**
- Updating your server.js file to "set" the "view engine" setting to use the value "ejs" (before your route definitions): **app.set('view engine', 'ejs');**
- Use the syntax **res.render("")** instead of **res.sendFile("")** in our route definitions**.** For example: **res.render("home")**, instead of **res.sendFile(__dirname + "/views/home.html").**

Part 3: Adding / Serving Static .ejs Files

Now that we have our primary "main" css file in place, and our server.js is configured to use ".ejs" files, we can focus on creating the static "views" for our application. These will be: **home.ejs** ("/"), **about.ejs** ("/about") and **404.ejs** (no matching route). These must be created according to the following specifications:

NOTE: Before you begin, do not forget to mark the "public" folder as "static", ie: **app.use(express.static('public'));** in your server.js file

<u>File: **/views/home.ejs**</u>

- Must reference "/css/main.css" (ie: compiled tailwindCSS) and include the correct "data-theme" attribute on the <html> element, if using a theme.
- Must have a <title> property stating something like "Climate Solutions"
- Have an element with class "container mx-auto", containing:
  - A "hero" daisyUI component featuring some text inviting users to explore the Projects and a link styled as a "btn" that links to "/solutions/projects"
  - A responsive grid system containing **3 columns**, each containing a "card" component featuring one of the items from your project (**hard-coded** in the html). Each card must contain:
    - An image from the project's "feature_img_url"
    - The project "title"
    - A summary from the project's "summary_short"
    - A link styled as a "btn" that links to "/solutions/projects/**id**" where **id** is the number of the project in the card, ie "/solutions/projects/2" for project 2
- Sent using the syntax **res.render("home");** from the **GET "/"** route in our server.js

File: **/views/about.ejs**

This file should follow the same layout as **views/home.ejs**, ie: reference main.css and have an appropriate <title> property

Additionally, this view should feature:

- an element with class "container mx-auto", containing:
    - A "hero" daisyUI component containing the header "About" with some additional text, ie "All about me", etc.
    - A responsive grid system containing **2 columns**:
        - The left column should show an image that you like / represents you.
        - The right column should be a short blurb about yourself (hobbies, courses you're taking, etc).
- Sent using the syntax **res.render("about");** from the **GET "/about"** route in our server.js.

File: **/views/404.ejs**

Once again, this file should follow the same layout as **views/home.ejs**, ie: reference main.css and have an appropriate <title> property

Additionally, this view should feature some kind of 404 message / image for the user. The sample uses a "hero" daisyUI component

- Finally, it should be sent using the syntax **res.status(404).render("404");** when **no route is matched / found** in our server.js.

Part 4: Creating a common "Navbar"

Since we are using a template engine that supports "partial" views, we should take this opportunity to create some common HTML, used across all pages, into a "partial" view. In our case, this is the navbar:

- In the "views" directory, create a new folder called "partials"
- Within the "partials" folder, create a new file: navbar.ejs

The contents of the file should feature **only the following content** (ie: no <html>, <head>, <body> elements, etc.)

- A **responsive** navbar with the following items:
    - "Climate Solutions" (or something similar) as the large text (left) which links to "/"
    - Link to "/about" with text "About"
    - Dropdown with Label / Summary "Sector"
        - The items in this dropdown should be links to **5 sectors** that are available in your dataset in the form: <a href="/solutions/projects?sector=**someSector**">**someSector**</a>

Once this is complete, you can place the navbar code as the first content within the <body> elements of your other 3 .ejs views, with the syntax: **<%- include('partials/navbar') %>**

Part 5: Adding / Serving Dynamic .ejs Files

At the moment, we still have multiple routes that only send data back to the client, ie: "/solutions/projects/", "/solutions/projects/id-demo" and "/solutions/projects/sector-demo". Since we are using a "template engine" (EJS) we can now send the data back as rendered HTML. Similarly, we can update our route definitions to use "route parameters" and "query parameters" to allow the user to view a range of projects by id or sector.

To begin, **comment** the route definitions for "/solutions/projects", "/solutions/projects/id-demo", and "/solutions/projects/sector-demo". We will delete these later, but will require some of their code for the upcoming routes.

Next, create the following routes / views:

GET "/solutions/projects/:id"

First, create the view "project.ejs" file within the views folder.

As a starting point for the HTML in this file, you can copy / paste the HTML from the existing view "404.ejs". Next, update the header ("hero" element) text by removing the content inside the <h1> and <p> elements (we will dynamically add these later).

Once this is complete, create a new GET "/solutions/projects/:id" route within your server.js

- Begin by copying the contents of your "/solutions/projects/id-demo" route.
- Instead of getting a fixed project id (ie: getProjectById(25)). Use the "id" route parameter instead
- Change the line **res.send(project);** (assuming that "project" is the object from your getProjectById() function) to: **res.render("project", {project: project});**
- Change the line **res.send(err);** (assuming that "err" is the error message from your getProjectById() function) to **res.status(404).render("404", {message: err});**

This will ensure that the "project" view is rendered with the project data stored in a "project" variable.

Before we test the server again, update your "project.ejs" to include the **project.title** in the <h1> element within your "hero" element, using the EJS Syntax: **<%= project.title %>**

With the server running, if you visit the route http://localhost:8080/solutions/projects/1, you should see a page with the text "Abandoned Farmland Restoration". Similarly, if you visit http://localhost:8080/solutions/projects/2, you should see a page with the text "Alternative Cement".

Finally, we should ensure that the following data from the "project" object is also rendered on the page – (see: https://wptf-a2-fall-2025.vercel.app/solutions/projects/1). This will include:

- An updated header ("hero" element) that also shows a blurb within the <p> element informing the user that they're viewing information for that particular project, ie "Below, you will find detailed information about the project: …"
- An image showing the image (using "feature_img_url") for the project
- A short intro ("intro_short") of the project
- The impact ("impact") of the project
- A link to find more information about the project (using "original_source_url")
- A quote from the url: "https://dummyjson.com/quotes/random" obtained by making an AJAX request when the page is loaded, ie: in the callback function for the "DOMContentLoaded" event:

```
<script>
  document.addEventListener("DOMContentLoaded", ()=>{

    /* TODO: "fetch" the data at: https://dummyjson.com/quotes/random and update an element in the DOM
with the "quote" and "author" */

  });
</script>
```

- A link with the following properties: href="#" onclick="history.back(); return false;" which serves as a "back" or "return" button

GET "/solutions/projects/"

Like with the above "/solutions/projects/:id" route, we will begin with creating the .ejs file first, ie; "**projects.ejs**" and copy / pasting the HTML from the existing view "404.ejs".

Once this is complete, create a new GET "/solutions/projects/" route within your server.js

- Begin by copying the contents of your "/solutions/projects/sector-demo" route.
- Instead of getting a list of fixed projects (ie: getProjectsBySector("transportation")). Use the "sector" query parameter instead
- Change the line **res.send(projects);** (assuming that "projects" is the list of objects from your getProjectsBySector() function) to: **res.render("projects", {projects: projects});**
- Change the line **res.send(err);** (assuming that "err" is the error message from your getProjectsBySector() function) to **res.status(404).render("404", {message: err});**

This will ensure that the "projects" view is rendered with the project data stored in a "projects" variable.

Before we test the server again, update your "projects.ejs" to include show the list of products (in JSON format) below your "hero", using the EJS Syntax: **<%= JSON.stringify(projects) %>**

Also, since the heading ("hero" element) still contains the static "404" text, we should update it by adding some hard-coded links to specific sectors (ie "Industry", "Transporation", etc) – (see: https://wptf-a2-fall-2025.vercel.app/solutions/projects?sector=Transportation)

Once this is working, we should ensure that the following data from the "projects" object is correctly rendered on the page – (see: https://wptf-a2-fall-2025.vercel.app/solutions/projects?sector=Transportation).

This will involve replacing the **<%= JSON.stringify(projects) %>** with a table featuring the following data in each row:

HINT: See "Iterating over Collections" for help generating the <tr>…</tr> elements for each Project set in the "projects" array

- The "title" of the project
- The "sector" of the project, which links to "/solutions/projects?sector=**sector**" where **sector** is the sector value of the project, ie: "Transportation" (consider styling this link as a button)
- The short summary ("summary_short") of the project
- A link with the text "details" that links to "/solutions/projects/**id**" where **id** is the "id" value for the project (consider styling this link as a button)

Once this is complete, you should see all your projects for a given sector rendered correctly in a table. Additionally, clicking on the "details" link should correctly redirect the user to find more information about a specific project!

We just have a couple potential issues – if the user visits the "/solutions/projects" route without a query parameter, we will see a 404 error, since no projects will be returned without a query parameter. Also, if the user enters a query parameter to specify a sector that doesn't exist, ie: "/solutions/projects?sector=unknown", then nothing is shown to the user.

To solve this, we must update the logic in our server.js for our new "/solutions/projects" route:

- We must check to see if there is a req.query.sector query parameter, ie: if(req.query.sector){ … }
    - If there is a req.query.sector query parameter, fetch the products using your getProjectsBySector() function as before
        - If there is at least 1 product in the array, return the data
        - If there are no products in the array, render a 404 view with the message: "No projects found for sector: **SECTOR**". Where, **SECTOR** is the value of the req.query.sector query parameter
    - If there is not a req.query.sector query parameter, fetch the products using your getAllProjects() function and return the data

Part 6: Updating 404.ejs to show errors

You will notice that in our above dynamic routes, we called **res.status(404).render("404", {message: …});**. This allows us to send a specific message to our 404 page, making it more specific for the user. As a final task, we should update our **404.ejs** file to render this message, ie:

- In the <p> element, replace any text that you have with **<%= message %>** (you can also keep some static text in there as well – it' sup to you)

- Ensure that anywhere in your server.js file where you render "404" *without* a message, ie: **res.status(404).render("404");** it is updated to show a message, ie: **res.status(404).render("404", {message: "I'm sorry, we're unable to find what you're looking for"});**

Part 7: Deploying your Site

Finally, once you have tested your site locally and are happy with it, it's time to publish it online.

Check the "Vercel Guide" for more information.

Assignment Submission:

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

```
/********************************************************************************
*  WEB322 – Assignment 02
*
*  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: ______________
*
*  Published URL: ___________________________________________________________
*
********************************************************************************/
```

- Compress (.zip) your assignment folder and submit the .zip file to My.Seneca under **Assignments** -> **Assignment** 2

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.