WEB322 Assignment 2

Submission Deadline:

Friday, November 7 – 11:59 PM

Assessment Weight:

10% 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, status codes and static content (css). We will then build upon that by refactoring our code to use the EJS template engine in order to render our data (instead of sending the JSON back to the client). Additionally, we will incorporate a random quote from the DummyJSON API

NOTE: Please refer the sample: https://web322-f25-a2.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.

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 the "tailwindcss" command as a "devDependency"
- Installing "@tailwindcss/typography" and "daisyui"
- Initializing tailwindcss
- Creating a "tailwind.css" file in /public/css
- Editing tailwind.config.js to ensure your .html files are watched during the build and "daisyui" / "@tailwindcss/typography" are added as plugins. 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: Adding .html Files

Now that we have our primary "main" css file in place, we can focus on creating the "views" for our application. At the moment, this is **home.html** ("/"), **about.html** ("/about") and **404.html** (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

File: **views/home.html**

- Must reference "/css/main.css" (ie: compiled tailwindCSS)
- Must have a <title> property stating something like "Climate Solutions"
- Include 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 **3 sectors** that are available in your dataset in the form: **someSector**
- 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

File: **views/about.html**

This file should follow the same layout as **views/home.html**, ie: reference main.css, have a <title> property and identical navbar. However, the navbar must have the text "About" highlighted by using the "active" class, ie:

About

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).

File: **views/404.html**

Once again, this file should follow the same layout as **views/home.html**, ie: reference main.css, have a <title> property and identical navbar.

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

Part 3: Updating server.js

To support dynamic routes, status codes and our custom 404 page, we must make the following changes to our server.js code from Assignment 1:

- Update the "/" route to respond with the "/views/home.html" file
- Add an "/about" route that responds with "/views/about.html" file
- Update the "/solutions/projects" route such that:
  - If there is a "sector" query parameter present, respond with Project data for that sector, ie "/solutions/projects?sector=industry" will respond with all projects in your collection with the "industry" sector
  - If there is not a "sector" query parameter present, respond with all of the unfiltered Project data
  - If any errors occurred, return the error message with the "404" status code
- Update the "/solutions/projects/id-demo" route such that:
  - Instead of returning the hard-coded Project from assignment 1, it should instead return the Project with an "id" value that matches the value after "/solutions/projects/". For example, "/solutions/projects/**2**" should return the Project with id **2**, etc.
  - If any errors occurred, return the error message with the "404" status code
- Delete the "/solutions/projects/sector-demo" route – we no longer need this one, since "/solutions/projects" now supports the "sector" query parameter

Add support for a custom "404 error". However, instead of returning text, respond with the 404 status code and the "/views/404.html" file

Part 4: Configuring EJS & Refactoring Existing Views

The first step will involve correctly installing ejs and configuring it in our server.js file, ie:

- Using npm to "install" the ejs package
- Updating your server.js file to "set" the "view engine" setting to use the value: "ejs"

Once this is complete, we can now change all of our ".html" files (ie: "**home.html**", "**about.html**", "**404.html**") to use the .ejs extension instead. Unfortunately, this means that any code that we have in our server.js to "send" the html files, ie:

res.sendFile(path.join(__dirname, "/views/home.html"));

will no longer work (since the file should now be "home.ejs"). To remedy this, change all of your "**res.sendFile()**" functions to "**res.render()**". For example, in the above case (for home.html) the new code would be:

res.render("home");

If you test your server now, you should not notice any changes in the browser (ie: the "about", "home" and "404" pages are all rendering correctly).

However, changing the filenames from .html to .ejs means that our code to "build" the main.css file using the command: "**npm run tw:build**" results in the following message:

"warn - No utility classes were detected in your source files. If this is unexpected, double-check the `content` option in your Tailwind CSS configuration."

This is because in our tailwind configuration, we're looking for .html files. As a result, the build process does not find any utility classes and does not correctly generate our main.css file.
To fix this, we will change the line in **tailwind.config.js**:

content: [`./views/*.html`], // all .html files

to instead read:

content: [`./views/**/*.ejs`], // all .ejs files

This will ensure that all .ejs files in the "views" folder (including sub-directories) will be found during our tailwind build step.

Part 5: Partial View – "navbar.ejs"

Since we are using a template engine that supports "partial" views, we should take this opportunity to move 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

Since every one of our view files share the same navbar structure, copy the complete navbar code from one of your views (ie: "home.ejs") – in the demo, this is the ** ... **element. Paste this code in your "navbar.ejs" file.

Once this is complete, you can replace the navbar code in your views with the include statement for the "navbar" partial:

<%- include('partials/navbar') %>

You should be able to run your server now and see that it is once again running as before (note: you may have to run npm run tw:build). However, you will notice that the navbar element for "about" is not highlighting correctly.

To correct this, we should add a "page" parameter to our partial view code that matches the link in the navbar that we wish to highlight, ie:

<%- include('partials/navbar', {page: '/about'}) %>

for the "about" view.

NOTE: We must always include some value for "page", even if there's no corresponding link in the navbar (ie: we can use {page: ''} for 404, etc).

To make sure the correct element in the navbar is highlighted within the partial view, we need to update each of our <a>… elements to conditionally add the 'active' class depending on the value of the "page" parameter. For example:

About

becomes

<a class="<%= (page == "/about") ? 'active' : '' %>" href="/about">About

and similarly,

Industry

becomes

<a class="<%= (page == "/solutions/projects") ? 'active' : '' %>" href="/solutions/projects?sector=ind">Industry

Part 6: Rendering "Projects"

Currently, the "/solutions/projects" route still returns the JSON data only. For this assignment, we will update this so that it shows a table of project data instead (see: https://web322-f25-a2.vercel.app/solutions/projects). To begin, create a new "projects.ejs" file within the views folder.

As a starting point for the HTML in this file, you can copy / paste the HTML from an existing view such as "404.ejs". Next, change the header ("hero" element) text to something more appropriate (ie: "Projects") and ensure that the "navbar" partial uses {page:"/solutions/projects"}.

To begin testing this route, change the server.js code defining your GET "/solutions/projects" route so that it renders the new "projects.ejs" file with the data instead of sending it directly (assuming it's stored in the variable "projects"), ie:

change **res.send(projects);** to **res.render("projects", {projects: projects});**

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

Rendering the Table

Now that the view is rendering with the "projects" data, we must display it in a table with 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)

Updating the header ("hero" element)

When testing your site, you should now see all of your project data rendered in a table! However, the heading ("hero" element) is still a little plain. To fix this, add some hard-coded links to filter your table by specific sectors (ie "Industry", "Transporation", etc) – (see: https://web322-f25-a2.vercel.app/solutions/projects)

Part 7: Rendering a "Project"

Currently, the "/solutions/projects/**id**" (where **id** matches a specific project), still renders the JSON formatted data. As in step 3, we must update this to show detailed project data instead (see: https://web322-f25-a2.vercel.app/solutions/projects/1). To begin, create a new "project.ejs" file within the views folder.

As a starting point for the HTML in this file, you can copy / paste the HTML from an existing view such as "about.ejs". Next, update the header ("hero" element) text by removing the content inside the <h1> and <p> elements (we will dynamically add these later) and ensure that the "navbar" partial uses {page:""} (since there's no matching page in the navbar).

To begin testing this route, change the server.js code defining your GET "/solutions/projects/:id" route so that it renders the new "project.ejs" file with the data instead of sending it directly (assuming it's stored in the variable "project"), ie:

change **res.send(project);** to **res.render("project", {project: project});**

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

Rendering the "Project" Data

If we test the server now, we will see that each individual project renders the same page without any specific project data. To fix this, ensure that the following data from the "project" object is rendered on the page – (see: https://web322-f25-a2.vercel.app/solutions/projects/1). This will include:

- An updated header ("hero" element) that shows the "title" of the project and a blurb 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

Part 8: Updating the Navbar & "404" view

At this point, most of the updates to the site have been completed – you should now be able to view the list of projects in a table, featuring data and links to individual projects, which are also rendered as HTML. However, there are some usability tweaks that we should add, including:

Updating the Navbar

Since we no longer require the "Sector" dropdown in the navbar, it can be removed. Instead, add a "View Projects" menu item that links to "/solutions/projects" *before* the "About" menu item. Also, do not forget to dynamically add the "active" class:

```
<li><a class="<%= (page == "/solutions/projects") ? 'active' : '' %>" href="/solutions/projects">View Projects</a></li>
```

NOTE: Do not forget to update both the regular and *responsive* navbar, as these links are duplicated.

Updating the "404" view (404.ejs)

There are a number of situations where it is appropriate to show a 404 error, ie: when projects with a specific sector, or id aren't found, or a route hasn't been defined. Because of this, it makes sense to show a different 404 message to the user depending on the type of error they have encountered. To achieve this, we should render the 404 view with a "message" property. For example: instead of

```
res.status(404).render("404");
```

use something like:

res.status(404).render("404", {message: "I'm sorry, we're unable to find what you're looking for"});

Now, in your "404.ejs" file, you can reference the "message" property using **<%= message %>**.

Finally, once this is complete, be sure to render the "404" error with an appropriate error message in the following situations:

- No projects found for a matching sector – (see: https://web322-f25-a2.vercel.app/solutions/projects?sector=asdf)
- No projects found for a specific id – (see: https://web322-f25-a2.vercel.app/solutions/projects/99)
- No view matched for a specific route – (see: https://web322-f25-a2.vercel.app/nope)

Part 9: Updating your Deployment

Finally, once you have tested your site locally and are happy with it, update your deployed site by pushing your latest changes to GitHub.

IMPORTANT NOTE: Now that we are using a template engine (ejs), do not forget to add the following code to your server.js file to ensure that your code runs on Vercel:

```
app.set('views', __dirname + '/views');
```

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: _____
*
*********************************************************************************/
```

- Compress (.zip) your web322-app (after removing node_modules folder).
- Get your hosted website's Vercel Public URL (Refer "How to get public Vercel URL" video in Notion Docs resources)
- Video 1: Website Demonstration (of website hosted on Vercel – no localhost)
    - Create a website demonstration screen recording **(minimum 3 mins, maximum 6 mins)** that starts by showing your website **on Vercel** in action. Begin by stating your name and briefly introducing your website, then systematically demonstrate all core functionality including user interactions, key features, navigation flow, and error handling scenarios. Ensure your demonstration covers all assignment requirements by showing the website working correctly with various inputs and edge cases. Maintain clear screen capture with a **minimum resolution of 1080p with audible narration** explaining each action as you perform it, ensuring the demonstration flows logically through your website's capabilities to prove that all required programming concepts function properly in practice.
- Video 2: Code Explanation
    - Create a comprehensive code walkthrough recording **(minimum 7 mins, maximum 10 mins)** where you systematically explain your implementation by showing your actual source code. Start by introducing yourself and providing an overview of your project structure and file organization, then walk through each required component specified in the assignment: demonstrate your use of language features, explain your design decisions, show how different concepts are implemented, highlight key functions and data structures, and point out how you handled challenging aspects of the assignment. Maintain clear screen capture with a **minimum resolution of 1080p with audible narration** while providing clear explanations that demonstrate understanding rather than just reading code aloud.

Check Notion Docs for resources on:

- How to do a proper app demonstration and code explanation for the submission videos
- How to upload videos to YouTube and set them as "unlisted"

Submit the following:

- **Project zip file** (after removing node_modules folder)
- **Hosted Website's Vercel Public URL** (Refer "How to get public Vercel URL" video in Notion Docs resources)
- **Video 1: Website Demonstration YouTube Video URL**
- **Video 2: Code Explanation YouTube Video URL**
    - upload the videos to YouTube, set them as "unlisted"
    - Share the links 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).**

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.