WEB322 Assignment 4

Submission Deadline:

Friday, March 8th, 2024 @ 11:59 PM

Assessment Weight:

9% of your final course Grade

Objective:

Build upon Assignment 3 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 "Quotable" API

If you require a *clean version* of Assignment 3 to begin this assignment, please email your professor.

NOTE: Please reference the sample: https://wptf-a4-sample.cyclic.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: Configuring EJS & Refactoring Existing Views

Since the major focus of this assignment is using EJS, the first step will involve correctly installing it 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 2: 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 ** ...
** elements. 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,

Technic

becomes

<a class="<%= (page == "/lego/sets") ? 'active' : '' %>" href="/lego/sets?theme=technic">Technic

Part 3: Rendering "Sets"

Currently, the "/lego/sets" route still returns the JSON data only. For this assignment, we will update this so that it shows a table of Lego set data instead (see: https://wptf-a4-sample.cyclic.app/lego/sets). To begin, create a new "sets.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: "Collection") and ensure that the "navbar" partial uses {page:"/lego/sets"}.

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

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

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

Rendering the Table

Now that the view is rendering with the "sets" 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 Lego set in the "sets" array

- An image showing the image (using "img_url") for the set (consider using an Avatar)
- The "name" of the set
- The "theme" of the set, which links to "/lego/sets?theme=**theme**" where **theme** is the theme of the set, ie: "technic" (consider styling this link as a button)
- The "year" the set was released
- The "num_parts" contained in the set
- A link with the text "details" that links to "/lego/sets/**set_num**" where **set_num** is the "set_num" value for the set (consider styling this link as a button)

Updating the header ("hero" element)

When testing your site, you should now see all of your Lego set 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 sets (ie "Technic", "Classic Town", etc) – (see: https://wptf-a4-sample.cyclic.app/lego/sets)

Part 4: Rendering a "Set"

Currently, the "/lego/set/**set_num**" (where **set_num** matches a specific set), still renders the JSON formatted data. As in step 3, we must update this to show detailed Lego set data instead (see: https://wptf-a4-sample.cyclic.app/lego/sets/001-1). To begin, create a new "set.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, change the header ("hero" element) text to contain empty <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 "/lego/sets/:num" route so that it renders the new "set.ejs" file with the data instead of sending it directly (assuming it's stored in the variable "legoSet"), ie:

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

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

Rendering the "Set" Data

If we test the server now, we will see that each individual set renders the same page without any specific set data. To fix this, ensure that the following data from the "set" object is rendered on the page – (see: https://wptf-a4-sample.cyclic.app/lego/sets/001-1)

- An updated header ("hero" element) that shows the "name" of the set and a blurb informing the user that they're viewing information for that particular set, ie "Below, you will find detailed information about the set: …"
- An image showing the image (using "img_url") for the set
- The "name" of the set
- The "year" the set was released
- The "theme" of the set
- The "num_parts" contained in the set
- A quote from the url: " https://quotable.io/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://quotable.io/random and update an element in the DOM with the "content" and "author" */

    });
  </script>
  ```

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

Part 5: 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 sets in your Lego collection in a table, featuring images and links to individual sets, 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 "Theme" dropdown in the navbar, it can be removed. Instead, add a "View Collection" menu item that links to "/lego/sets" *before* the "About" menu item. Also, do not forget to dynamically add the "active" class:

```
<li><a class="<%= (page == "/lego/sets") ? 'active' : '' %>" href="/lego/sets">View Collection</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 sets with a specific theme, 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 (ie: the message returned from a rejected promise when attempting to find a specific Lego set) in the following situations:

- No Sets found for a matching theme – (see: https://wptf-a4-sample.cyclic.app/lego/sets?theme=asdf)
- No Sets found for a specific set num – (see: https://wptf-a4-sample.cyclic.app/lego/sets/asdf)
- No view matched for a specific route – (see: https://wptf-a4-sample.cyclic.app/asdf)

Part 6: 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.

Assignment Submission:

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

```
/*********************************************************************************
*  WEB322 – Assignment 04
*
*  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** 4

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.