WEB322 Assignment 4

Assessment Weight:

9% of your final course Grade

Objective:

Build upon the code created in Assignment 3 by incorporating the Handlebars view engine to render our JSON data visually in the browser using .hbs views and layouts. Additionally, update our blog-service module to support additional functionality.

NOTE: If you are unable to start this assignment because Assignment 3 was incomplete - email your professor for a clean version of the Assignment 3 files to start from (effectively removing any custom CSS or text added to your solution).

Part 1: Getting Express Handlebars & Updating your views

Step 1: Install & configure express-handlebars

- Use npm to install the "express-handlebars" module
- Wire up your server.js file to use the new "express-handlebars" module, i.e.
    - "require" it as exphbs
    - add the app.engine() code using exphbs.engine({ … }) and the "extname" property as ".hbs" (See the Week 6 Notes)
    - call app.set() to specify the 'view engine' (See the Week 6 Notes)
- Inside the "views" folder, create a "layouts" folder

Step 2: Create the "default layout" & refactor about.html to use .hbs

- In the "layouts" directory, create a "main.hbs" file (this is our "default layout")
- Copy all the content of the "about.html" file and paste it into "main.hbs"
    - **Quick Note:** if your site.css link looks like this href="css/site.css", it must be *modified* to use a leading "/", ie href="/css/site.css"
- Next, in your main.hbs file, remove all content **INSIDE** (not including) the single … element and replace it with {{{body}}}
- Change the <title></title> attribute to remove "About" and change it to include your student name, ie "Homer Simpson's Blog"
- Once this is done, rename about.html to about.hbs

- Inside about.hbs, remove all content **EXCEPT** what is INSIDE the single … element (this should leave a single … element containing two "columns", ie elements with class "col-md- …" and their contents)
- In your server.js file, change the GET route for "/about" to "render" the "about" view, instead of sending about.html
- Test your server - you shouldn't see any changes. This means that your default layout ("main.hbs"), "about.hbs" and server.js files are working correctly with the express-handlebars module.

Step 3: Update the remaining "addPost" file to use .hbs

- Follow the same procedure that was used for "about.html", for the "addPost.html" file, i.e.
    - Rename the .html file to .hbs
    - Delete all content **EXCEPT** what is INSIDE the single … element
    - Modify the corresponding GET route (i.e. "/post/add") to "**res.render**" the appropriate .hbs file, *instead* of using res.sendFile
- Test your server - you shouldn't see any changes, ***except*** for the fact that the "Add Post" menu item is no longer highlighted when we change routes (only "About" remains highlighted, since it is the only menu item within our main.hbs "default layout" with the class "active"

Step 4: Fixing the Navigation Bar to Show the correct "active" item

- To fix the issue we created by placing our navigation bar in our "default" layout, we need to make some small updates, including adding the following middleware function ***above*** your routes in server.js:

```
app.use(function(req,res,next){
  let route = req.path.substring(1);
  app.locals.activeRoute = "/" + (isNaN(route.split('/')[1]) ? route.replace(/\/(?!.*)/, "") : route.replace(/\/(.*)/, ""));
  app.locals.viewingCategory = req.query.category;
  next();
});
```

  This will add the property "activeRoute" to "app.locals" whenever the route changes, i.e. if our route is "/blog/5", the app.locals.activeRoute value will be "/blog ". Also, if the blog is currently viewing a category, that category will be set in "app.locals".

- Next, we must use the following handlebars custom "helper" (See the Week 6 notes for adding custom "helpers")"

```
navLink: function(url, options){
  return '<li' +
    ((url == app.locals.activeRoute) ? ' class="active" ' : '') +
    '><a href="' + url + '">' + options.fn(this) + '</a></li>';
}
```

- This basically allows us to replace all of our existing navbar links, i.e. About with code that looks like this {{#navLink "/about"}}About{{/navLink}}. The benefit here is that the helper will automatically render the correct <li> element add the class "active" if app.locals.activeRoute matches the provided url, ie "/about"
- Next, while we're adding custom "helpers" let's add one more that we will need later:

```
equal: function (lvalue, rvalue, options) {
  if (arguments.length < 3)
    throw new Error("Handlebars Helper equal needs 2 parameters");
  if (lvalue != rvalue) {
    return options.inverse(this);
  } else {
    return options.fn(this);
  }
}
```

  This helper will give us the ability to evaluate conditions for equality, ie {{#equals "a" "a"}} … {{/equals}} will render the contents, since "a" equals "a". It's exactly like the "if" helper, but with the added benefit of evaluating a simple expression for equality

- Now that our helpers are in place, update ***all the navbar links*** in main.hbs to use the new helper, for example:
  - About will become {{#navLink "/about"}}About{{/navLink}}
- Test the server again - you should see that the correct menu items are highlighted as you navigate between views

Part 2: Updating the Posts Route & Adding a View

Rather than simply outputting a list of posts using res.json, it would be much better to actually render the data in a table that allows us to filter the list using our existing req.params code.

Step 1: Creating a simple "Posts" list & updating server.js

- First, add a file "posts.hbs" in the "views" directory

- Inside the newly created "posts.hbs" view, add the html:

```
<div class="row">
  <div class="col-md-12">
    <h2>Posts</h2>
    <hr />

    <p>TODO: render a list of all post title values here</p>

  </div>
</div>
```

- Replace the <p> element (containing the TODO message) with code to iterate over **each post** and simply render their title properties (you may assume that there will be a "posts" array (see below).
- Once this is done, update your GET "/posts" route according to the following specification
    - Every time you would have used res.json(data), modify it to instead use res.render("posts", {posts: data})
    - Every time you would have used res.json({message: "no results"}) - i.e. when the promise has an error (ie in .catch()), modify your code to use res.render("posts", {message: "no results"});

Step 2: Building the Table & Displaying the error "message"

- Update the posts.hbs file to render all of the data in a table instead of a list, using the bootstrap classes: "table-responsive" (for the containing the table) and "table" (for the table itself)
    - The table must consist of 5 columns with the headings: **Post ID**, **Title**, **Post Date**, **Category** and **Published**
    - Additionally, the Category values in the Category column must link to /posts?category=**category** where **category** is the category for the post for that row
- Beneath … element, add the following code that will conditionally display the "message" only if there are no posts (**HINT**: #unless posts)

```
<div class="col-md-12 text-center">
    <strong>{{message}}</strong>
</div>
```

This will allow us to correctly show the error message from the .catch() in our route

Part 3: Updating the Categories Route & Adding a View

Now that we have the "Post" data rendering correctly in the browser, we can use the same pattern to render the "Categories" data in a table:

Step 1: Creating a simple "Categories" list & updating server.js

- First, add a file "categories.hbs" in the "views" directory
- Inside the newly created "categories.hbs" view, add the html:

```
<div class="row">
  <div class="col-md-12">
    <h2>Categories</h2>
    <hr />

    <p>TODO: render a list of all category id's and categories here</p>

  </div>
</div>
```

- Replace the <p> element (containing the TODO message) with code to iterate over **each category** and simply render their id and category values (you may assume that there will be a "categories" array (see below).
- Once this is done, update your GET "/categories" route according to the following specification
    - Instead of using res.json(data), modify it to instead use res.render("categories", {categories: data});
    - When the promise has an error (ie in .catch()), modify your code to use res.render("categories", {message: "no results"});
- Test the Server

Step 2: Building the Table & Displaying the error "message"

- Update the categories.hbs file to render all of the data in a table, using the bootstrap classes: "table-responsive" (for the containing the table) and "table" (for the table itself)
    - The table must consist of 2 columns with the headings: **Category ID** and **Category Name**
    - Additionally, if you click on either the cateogry id, or the category name, you'll be redirected to /posts?category=X, where X is the category id for the category that was clicked (see above link for example)

- Beneath … element, add the following code that will conditionally display the "message" only if there are no categories (**HINT**: #unless categories)

  ```
  <div class="col-md-12 text-center">
      <strong>{{message}}</strong>
  </div>
  ```

  This will allow us to correctly show the error message from the .catch() in our route

Part 4: Updating the Blog Route & Adding a View

Our next JSON-to-Handlebars conversion task is related to showing the actual Blog route. This one is slightly more complicated, as it will involve determining which is the "latest" blog post and displaying that, while also displaying links to the other blog posts and categories in a sidebar.

Step 1: Adding a new "Helper" to handle unsafe HTML in posts

- Before we start creating our new view, it is important to consider the possibility of unsafe <script></script> tags or other JavaScript code in the html for the blog post. If we want to be able to render html within the blog post, we must handle this situation. For this assignment, we will use a custom helper called **safeHTML** that removes unwanted JavaScript code from our post body string by using a custom package: **strip-js** (https://www.npmjs.com/package/strip-js)
  - First, run the command **npm i strip-js**
  - "require" it using the line: **const stripJs = require('strip-js');**
  - Add the custom helper:

    ```
    safeHTML: function(context){
      return stripJs(context);
    }
    ```

  - You can now use this helper with the syntax:

    {{#safeHTML **someString**}}{{/safeHTML}}

    where **someString** may be your post **body**, for example

Step 2: Updating the blog-service.js module

- This view will be capable of filtering posts by Category. However, we currently do not have a function that produces posts that are both **published** and **filtered by Category**. As a result, we must add a new blog-service function called **getPublishedPostsByCategory**(**category**)
    - This function works exactly as **getPublishedPosts()** except that in addition to filtering by "post.published == true", you must also include category in the filter, i.e. "**post.published == true && post.category == category**"

Step 3: Creating the "Blog" view & updating server.js

- First, add a file "blog.hbs" in the "views" directory
- Inside the newly created "blog.hbs" view, add the html from here (as a starting point): https://pat-crawford-sdds.netlify.app/shared/winter-2022/web322/A4/blog.hbs.txt
- Before we start editing our new template, let's first get the data in place so that you can test it as you go:
    - Open server.js and replace your current app.get("/blog") route with the code available here: https://pat-crawford-sdds.netlify.app/shared/winter-2022/web322/A4/get-blog.txt

      NOTE: this code assumes that you reference your blog-service.js using the variable **blogData**, i.e. const **blogData** = require("./blog-service");
- You should now be able to access information for:
    - The current blog post object using **data.post**
    - The current array of blog posts using **data.posts**
    - The current array of categories using **data.categories**
    - Potential error obtaining blog posts using **data.message**
    - Potential error obtaining categories using **data.categoriesMessage**
- With this information, update your newly created **blog.hbs** according to the following specification:
    - Update the <h2> element at the start of the <article> to show the current post title
    - Update the src attribute for "feature image" to show the current post featureImage
    - Replace the existing long "Lorem Ipsum" string (between the

 …

 elements) with the actual body of the current post using our new **#safeHTML helper** (see above)
    - Update the "Category:" value to show the current post category
    - Update the "Last Updated:" value to show the current post postdate
    - Update the Posts list to show real posts using **data.posts.** Additionally:
        - Each "href" value must link to "/blog/**id**?category={{../viewingCategory}}" where **id** is the **id** value for the post shown in the list. **NOTE:** ../viewingCategory will allow us to access the global "viewingCategory" value (set in our middleware function at the start of this assignment) and the "/blog" route will be created below.

        - The Posts list (including the <h4>Posts</h4> element) must not be visible if **data.posts** is undefined or empty (**HINT:** #if data.posts)
    - Update the Categories list to show real categories using **data.categories**. Additionally:
        - **NOTE:** Each "href" value must link to "/blog?category=**id**" where **id** is the **id** value for the category shown in the list
        - The Categories list (including the <h4>Categories</h4> element) must not be visible if **data.categories** undefined or empty (**HINT:** #if data.categories)
    - Finally, make sure that the entire <article>…</article> element is only visible if there is a post to show (**HINT:** #if data.post).
        - If there isn't a post to show, show **data.message** using either the following HTML:

            <h2>{{data.message}}</h2>

            Please try another Post / Category

          or something similar, if you prefer a different style or accompanying text

Part 5: Adding the Blog/:id Route

The last major piece of this assignment is to ensure that individual blog posts can be rendered using the same layout as the main blog page. However, instead of displaying the latest blog post available / per category, we must instead show a specific blog post (by **id**).

This can be accomplished by adding a new route with the code available here:
https://pat-crawford-sdds.netlify.app/shared/winter-2022/web322/A4/get-blog-id.txt

You will notice that its nearly identical to the app.get('/blog')… route, except instead of using the most recent blog post, we will instead use the blog post with the id obtained from the route parameter, id.

Part 6: Final Updates (setting PostDate, redirecting to /blog & 404.hbs)

To ensure that any new posts created will show up at the top of the blog, you must update the "**addPost**" method of your blog-service module to ensure that the "postDate" field is correctly set when a new blog post is created. This can be accomplished by:

- Adding the property "postDate" to your postData object **before** you push it to the "posts" array.
    - The value will be the current date formatted using the pattern YYYY-MM-DD (to match the existing format – NOTE: YYYY-M-D is fine as well, i.e. you don't need to worry about leading 0's)

Before we finish our assignment (if you haven't created a custom 404 page / converted it to handlebars yet), consider adding a "404.hbs" file that is rendered instead of a plain message. This will have the benefit of keeping the menu bar intact so that users can quickly get back to the content in the event of a 404 error.

Finally, as the last step before completing the assignment, update your default "/" route to **redirect** to "/blog" instead of "/about"

Part 7: Pushing to GitHub and Cyclic

Once you are satisfied with your application, push to GitHub and deploy it to Cyclic:

- Ensure that you have checked in your latest code using **git** (from within Visual Studio Code)
- Push commits to the same *private* **web322-app** GitHub repository either through the integrated terminal (**git push**) or through the button interface on Visual Studio Code (publish, sync, etc.)
- If set up correctly from Assignment 2, it will automatically be deployed to Cyclic but if there are any problems, follow the Cyclic Guide on web322.ca for more details on pushing to GitHub and linking your app to Cyclic for deployment
- **IMPORTANT NOTE:** Since we are using a **free** account on Cyclic, we are limited to only **3 apps**, so if you have been experimenting on Cyclic and have created 3 apps already, you must delete one. Once you have received a grade for Assignment 1, it is safe to delete this app (login to the Cyclic website, click on your app and then click **Advanced** and finally, **Delete App**).

- The "helloprof" GitHub account should already be added as a collaborator to your **web322-app** GitHub repository

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  Academic Policy.  No part
* of this assignment has been copied manually or electronically from any other source
* (including 3rd party web sites) or distributed to other students.
*
* Name: _____ Student ID: _____ Date: _____
*
```

```
* Cyclic Web App URL: _____
*
* GitHub Repository URL: _____
*
********************************************************************************/
```

- Compress (.zip) your web322-app folder and submit the .zip file to My.Seneca under **Assignments** -> **Assignment** 4

Important Note:

- Submitted assignments must run locally, i.e. start up errors causing the assignment/app to fail on startup will result in a **grade of zero (0)** for the assignment.