Module 2 Lab 2: Basic Full Stack Labs

Ryan M.K. Burke-Jimenez

Arizona State University

IFT 458

Professor Dinesh Sthapit

September 10th, 2023

Module 2 Lab 2: Basic Full Stack Labs

This lab is designed to implement a basic full stack environment onto a local host for testing and learning purposes.

# Environment Setup

Visual Studio Code was the environment used to set up the files for inspection. Upon creating a new directory for the project to be held, NPM was used to initialize the project and create a package.json file. NPM was also utilized to install express, body-parser, and ejs, all modules that are necessary for this assignment. After this, three main files were created for the demonstration: app.js (main file), index.ejs, and result.ejs. Index.ejs and result.ejs are both held in a separate folder named “views” that is held in the same directory as app.js. App.js utilizes Node.js while index.ejs and result.ejs utilize HTML (Figure 1). The project was subsequently committed and pushed onto GitHub for assessment. When the local server is operational on port 4000, inspecting the elements will show the corresponding HTML source code in the web browser, in this case, Google Chrome. The environment is now successfully established for further analysis.

A screenshot of a computer program

Description automatically generated

Figure - index.ejs, result.ejs, app.js

## Exercise 1: Analyzing .ejs Code vs. HTML Rendered in the Browser

The purpose of this exercise is to observe and analyze the difference between an .ejs document with a rendered .html document that is present in a browser’s element inspection.

## .ejs Code Analysis

There are two .ejs files that exist in this project, however, only one is relevant for inspection: result.ejs. Result.ejs contains JavaScript within the HTML, which is due to the files being .ejs rather than .html files. EJS is short for Embedded JavaScript and is a powerful way to generate HTML with JavaScript components. To embed JavaScript within the HTML file, it must be present between <%= variable %> and is between HTML element paragraphs with the result being <p>Variable: <%= variable%></p>. The HTML element <p></p> may be just for aesthetics where any other HTML element will suffice. The <%=%> structure is necessary for the .ejs file to understand that JavaScript is present in the code and will set variables to the given placeholders, in this case, various mathematical operations are variables, and the calculations are completed within the app.js file. The variables are set in result.ejs when index.ejs communicates with app.js through the post function (Figure 2). App.js requests information that is entered in the UI of index.ejs and with the inputs app.js can respond with the

A screenshot of a computer program

Description automatically generated

2. Calculations complete, response acknowledged. Render and results sent back to result.ejs. Variables set with results sent from app.js

1. “Calculate” in UI pressed, POST function requested. Index.ejs sends information to app.js.

Figure - Communication route for result.ejs

calculations when the “calculate” button in the UI is pressed. Once pressed, result.ejs is rendered and receives the variables from the POST function where the results are displayed in the UI.

### HTML Code Analysis

After careful analysis of the .ejs file, analysis of the rendered HTML file is next for observation. In the element inspection of the UI, the source code of result.ejs is present, however, no JavaScript is shown in the HTML file (Figure 3). In the HTML file, the results of the POST call from app.js are displayed but the JavaScript in result.ejs is not. The results being present rather than their variable names were to be expected, however, the <%=%> structure is not shown within the HTML file which is rather intriguing. This must mean that once the result.ejs file receives the result of the POST function, EJS knows to fully replace <%= variable%> with the result without any complication. Other than this, the .ejs and the HTML file are similar in structure and nothing else is modified.

A screenshot of a computer

Description automatically generated

Figure - HTML file inspection

Compare and Contrast. In structure, HTML and EJS are similar in that HTML can be generated in both file types without any issue. All syntax and elements used in HTML may be used in EJS, however, it is not reciprocal. EJS has the ability to hard code JavaScript into the source code while using HTML as the base code for the file. This can be a very powerful tool in the event that the user needs to output JavaScript while developing a website with HTML that has dynamic content as a function of the website (Ikechukwu, 2018). In the event that a developer needs to provide users with a frontend UI with the proper workings of the backend embedded into the file, EJS is the most beneficial way to accomplish this.

**Exercise 2: Arrays and Objects**

This exercise demonstrates the use of Arrays and Objects in Node.js and how to display this information in JavaScript located in the newly created *books.ejs* file. The books.ejs file was created in the views folder in the hierarchy and consists of HTML and JavaScript source code (Figure 4).

A screenshot of a computer

Description automatically generated

Figure - Books.ejs Template Source Code

In the app.js file, a GET and POST function were both added to acquire information from books.ejs that will be broadcasted on port 4000 (Figure 5). The object array “books” is also housed in the app.js file which stores the data that is inputted by the user on the webpage. Each value of the array is then a response in the POST function, where books.ejs displays the newly acquired information to the user on the UI.

A screenshot of a computer program

Description automatically generated

Figure - Updated app.js file

This information is then responded back to books.ejs in the POST function of app.js, where books.ejs then posts the information to the user on the UI (Figure 6). The modification I chose to implement in the front-end is to display information below the dropdown menus to show more fluidity in structure. There were no errors on runtime to report.

A computer screen with a white background

Description automatically generated

Figure - Webpage for books.ejs

**Exercise 3 – Constructors and DE structuring**

This exercise demonstrates the use of constructors and de-constructors in app.js as well as properly implementing two new EJS files; users.ejs and userinfo.ejs. Given the templates in the lab guideline, a new User function is the constructor of User objects through calling its own parameter to be set to its own variable, making it a constructor of a User object that does not return a value(Figure 7).

A screenshot of a computer

Description automatically generated

Figure - User Constructor + GET and POST functions

This User object is then sent implemented in the route handler of the new users.ejs file and a new GET server call was created to render the new users.ejs file. The parameter in the URL path for the users.ejs file to be rendered is /users. In the users.ejs file, the standard HTML template that was provided in the lab instructions were implemented as well, however, some modifications were made to accommodate the needed fields of the User objects (Figure 8). A computer screen shot of a computer

Description automatically generated

Figure - Users.ejs File

These parameters are displayed in the results webpage that is userinfo.ejs. Userinfo.ejs consists of a simple embedded script in the HTML file to display the User objects. The routing to this file is embedded in the post function, where the button for “Create User” is pressed then the user is automatically rerouted to the resulting userinfo.ejs file properly displaying the user to the screen (Figure 9).

A screenshot of a computer

Description automatically generated

Figure - Userinfo.ejs file + users and userinfo displayed in browser

**Exercise 4: Spread Operator**

This exercise is to demonstrate the use of a spread operator in the app.js file while creating a new fruits.ejs file to display the function on the frontend. Spread operators are used to quickly display data that is stored in an array. The POST function pushes each predetermined array element into another fruit array that is displayed in fruits.ejs. The GET function is routed to handle a call from the browser at /fruits and renders the fruits.ejs file when searched. The spread operator is seen in JavaScript in fruits.ejs, where each element of the array is quickly displayed on page creation. Figure 10 shows the UI and resulting array on display. A customization is displaying the list below the field and a header showing which exercise the demonstration is for.

A screenshot of a computer

Description automatically generated

Figure - Fruits.ejs + spread on UI

**Exercise 5: Promises**

Exercise 5 demonstrates how Node.js handles asynchronous functions through the use of promises. In this example, the async GET function is called when the webpage of /simulateAsync is searched in the browser. With the setTimeout call it simulates the process of a overactive server that may be experiencing callback lag when the request is queued in the stack. The async.ejs file holds another script that demonstrates a delay in fetching the relative webpage of /simulateAsyc and displays the text written in the GET function for /simulateAsync (Figure 11).

A screenshot of a computer screen

Description automatically generated

Figure - async.ejs + /simulateAsycn GET function

**Exercise 6: Async/Await**

The final exercise is to demonstrate how HTTP responses are handled with the use of Fetch and Axios. Fetch is an API that is used to fetching resources called for and handling HTTP requests where Axios is a third-party JavaScript library that accomplishes nearly the same thing as Fetch only that it is less complex and works for both front and backend applications. Axios works in the /makeRequest POST function in app.js when returning the URL from the form on the webpage and Fetch is used in the “submit” button request in the httpRequest.ejs file to retrieve the response of /makeRequest on button press (Figure 12). Before beginning, I had to implement another GET request to properly render httpRequest.ejs when called to by localhost:4000.

A screenshot of a computer program

Description automatically generated

Figure - Fetch + Axios Demonstration

When running the code provided in the lab, there was no response from the browser initially when prompted to search for the API that was also given in the lab instructions. Upon researching the possible issue, there were two variables by the name of “response” in the httpRequest.ejs file that were causing data flow errors. The first response stores the data element by ID “Response” which is an empty paragraph in HTML whereas the second response stores the Fetch call on /makeRequest. I elected to rename each variable responseElement for the element by ID and fetchedData for the Fetch call. There was another human error in the form of a typo on my part that still did not allow the data flow to run correctly, but after finding and fixing the typo, the script ran smoothly (Figure 13).

A screenshot of a computer

Description automatically generated

Figure - Result of running /makeRequest

References

Ikechukwu, L. (2020, August 13). Using EJS as a Template Engine in your Express App - Linda Ikechukwu - Medium. *Medium*. <https://medium.com/@Linda_Ikechukwu/https-medium-com-linda-ikechukwu-using-ejs-as-a-template-engine-in-your-express-app-cb3d82c15e17#:~:text=What%20is%20EJS%3F,interpolate%20(concatenate)%20strings%20effectively>.