Web Application Development (Level 5) Report

João Bota

Q14756803

**Part A – Develop a very simple REST API**

Firstly, I have started this task by installing the **Node libraries**, **mariadb** and importing modules such as **express** and **mysql.** Afterwards, I installed the database that was given by the Tutor. Then, I created a **GET Request** to look up all points of interest in a given region on the database, by querying “**SELECT \* FROM pointsofinterest WHERE region =?”**, and two **POST Request**, which added a new point of interest to the database by using the **API endpoint**, **“http:localhost:3000/create”**, using the query, **“INSERT INTO pointsofinterest ( ID, name, type, country, region, lon, lat, description, recommendations) VALUES (?,?,?,?,?,?,?,?,?)”**, and the other **POST Request**, enabled to recommend a point of interest by querying **“UPDATE pointsofinterest SET recommendations=recommendations+1 WHERE ID=?”**. Each request if there was an error, it should send back a **JSON** object containing an error property with a value of the error that occurred, on the other hand, if the operation was successful the call back parameter “**results”** will contain the results in JSON. In the **Part A**, I did not encounter any problems.

**Part B – Develop a simple AJAX-based JavaScript front-end**

To begin this task, I have created a **index.html** file, which is the main page and allows the user to search for all points of interest in a given region. The user should write the region on the **input** with the **id** (**regionInput**)and by clicking on the **SearchButton** it should appear an array of **JSON** objects as the result. In order to make this possible, I have created the **scripts.js** file which has **asynchronous** functions and **event listeners,** which manipulate the **HTML document** with the call back given by the parameter **results** in the **app.js**. All of this is possible because by using **JavaScript**, the page communicates with the **REST API** to find all the points of interest and parses the **JSON** objects and presents them to the user.

As asked, I created another HTML page which allows the user to enter the **point of interest** details and create a new **point of interest**. I have created **scriptPost.js** to separate the **AJAX functions** from the main **HTML** page.

Afterwards, I modified the code which processes the search results on the **script.js**, to create a **“Recommend”** button that sends an **AJAX POST request** to the **REST API** to allow the user to recommend the point of interest.

I had no problems with the completion of this task because I have some experience with **AJAX**.

**Part C – Adding simple error-checking**

This task was done previously in the **Part A**. I prefer to do the simple error-checking while I am building the **REST API**, so if the user leaves details of the point of interest in blank, an appropriate **HTTP** error code (500) is sent back and with **AJAX**, it displays an appropriate message to the user.

**Part D – Adding a map**

To start off, I have created a map in **scripts.js** with the name of “**map1**”, the latitude of the starting position is **50.908** and the longitude is **-1.4**. Afterwards, inside of the **index.html**, I have linked the Leaflet library as an external **JavaScript** file and the **Leaflet CSS** file. Then, I have created a div with the **id** of “**map1**” and with a width of **800px** and a height of **600px.** In the **scripts.js** file, I have written a marker which sets an icon on the starting position and by clicking on the marker it should appear the following message: “**My Location**”. Afterwards, as requested, I have updated the function **ajaxSearch**, to create a marker in the position of each region that resulted from the search that the user entered (**regionInput**), also, if the user clicks on the marker from the results, it should appear a popup containing the point of interest name (**region.name**) and the description of the point of interest (**region.description**).

Also, I have added the feature to allow the user to add a point of interest by clicking on the map at a particular location. This was made possible by creating an **event listener** which, when the user clicks on the map, it creates a marker (**marker2**), on the position that was clicked (**e.latlng.lat, e.latlng.lng**) and manipulates a **<div>** (**create**), previously created in the **HTML file**, to show a form and binds it to the “**marker2**” as a popup and displays a button (**CreateButton**) at the bottom of the map, that is directly connected to the **REST API** and **AJAX**, to proceed with the creation of the new point of interest. The task was challenging, because while moving forward with the tasks the application began to increase in complexity, which makes a simple mistake cause a lot of errors and slows down the creation of the final app. Most of the errors that I encountered during this task were typos and I had to update the code multiple times in order to have a greater logic on the app.

Part E – Logins and sessions

In this task, I have implemented a session-based login system. To start off, I have created a **<div>** with the **id** (**loginForm**), that enables the user to input his username and password. Afterwards, I have installed **cookie-parser** **library** and **express-session library**, then, I started setting up the modules. I did not use “**express-mysql-session**” because I was having problems setting up the module, after various attempts I could not store the data in a table. I have searched online and used as an alternative **the cookie-parser**. I have used authentication, so the user must enter a username and password to gain access to the site. In order to make authentication work, I have started by creating the **login route** which tests if the user entered a valid username and password that was previously created on the database inside **poi\_users**, and if the request was successful, it stores the username in a session variable (**req.session.user**). Inside of **scripts.js**, I have updated the code by creating an **asynchronous function (ajaxLogin)** which whenever the user successfully logged in, it displays an alert and manipulates the **<div> (“loginForm”)** to show the message, “Logged in as …” and displays a **logoutButton**, which on click, it runs the **asynchronous function (ajaxLogout)**, and displays an alert and manipulates the **<div> (“loginForm”)**, to show again the log in form. Also, I have added to **app.js** a route (**GET Login Route**), to obtain the currently logged in user. I have also updated the **app.js** so the user must be logged in to add a point of interest. I have created a function **(Middleware)**, that blocks the user from creating a new point of interest, by preventing the **POST request** from the **REST API,** which displays an alert to prompt the user to login and sends the appropriate **HTTP** error code (401) to the REST API.

Also, I have updated the **asynchronous function(ajaxCreate)**, to display an alert whenever the **HTTP error code (401)** emerged.

In my opinion, this was the most difficult part, due the lack of experience with sessions and authentication. I had to do a lot of research in order to understand better and create a logical app.#

**Part G – Improving your answer**

In order to create a well-structed Node application, I have used controllers and routes. I started by creating a folder (**routes**) and created the **pointsofinterest.js** and **login.js**. In both files, I have imported **express.Router** to export both routes to the main app (app.js), the **pointsofinterest.js** exports **pointofinterestRouter** and **login.js** exports **loginRouter**. In order to work properly, both **pointofinterestRouter** and **loginRouter** were imported to **app.js**.

Also, I have created a new file (**middleware.js**) which exports the variable “**Middleware**”, in order to make the code look cleaner.

I have created a new file (**connection.js**) which exports the settings to create and connect to the database to **app.js**

Finally, inside **app.js** and **connection.js** I required “**dotenv**” which allowed to separate secrets form the source code, **(DB\_HOST, DB\_USER, DB\_DBASE)**.

This final task was challenging due of the amount of code inside the file (**app.js**), after the task was completed the structure of the Node application looked a lot better.