COM518 AE1

REPORT

NAME: MUHAMMAD AHUB SAJID

ID: Q15586626

Table of Contents

INTRODUCTION

In this assignment, we were asked to develop a web application that would work alongside an SQLite database to allow users to look up information on places they may want to visit. We were required to use Node and Express as the back-end technology and a mixture of JavaScript and HTML to develop the front-end.

PART A

The first part of the assignment required us to develop a simple REST API using node and express. This API would allow users to look up all the POIs in a given region, add a new point of interest and recommend a point of interest.

Task 1 will allow the user to search for POIs in a given region from the database. To implement this, I had to create a route in the API that, when called, perform a SQL select query using the region given in the parameters of the request. A list of POIs is then returned to the server and if the list contains POIs then a 200 response code will be set to show a success otherwise a 500 response code is set to show an error has occurred.

Task 2 allows the user to add a new POI to the database. First I created a new route that performs an SQL insert query and uses the POST data from the http POST request to specify the specific values. The database returns the number of rows that changed and if this is one then a 200 response code is sent back to show a success otherwise a 400 response code is sent back to signify an error occurring. I have chosen to use a 400 code as it represents the server not carrying out a http request due to a client error which in this case may be blank values.

Task 3 is used to recommend a POI. Firstly I created a new route that took in an ID in the parameters and this ID is used to carry out an SQL update query to increment the recommendations column for the specified POI. The database returns the number of rows that changed and if this is one then a 200 response code is sent back to show a success otherwise a 404 response code is sent back to signify an error occurring. A 404 error represents a ‘Not Found’ error so in this case this will occur if the POI with the specified ID can’t be found.

PART B

The second section of the assignment tasks required us to develop a simple HTML and JavaScript front-end which communicates with the REST API using AJAX.

To implement the fourth task which was to look up all the POIs in a given region, I created a JavaScript page that would be used to carry this out. I created a function within this page that would take in the region that the user would like to search for. This function would have to be asynchronous as we want the results to be displayed without refreshing the page so we will wait for the response when we send the http request. This is because we are using AJAX and the easiest way to communicate with the server is using the fetch API. When we 'fetch' something in AJAX the return object is a promise object. A promise object quite literally "promises" to perform a certain task in the background. This means that when we 'fetch' something, straight away a promise object is returned; to be fulfilled or rejected in the future. Talking about the http request, this is what we are going to do first to retrieve the POIs from the database. As we created the route in the task one of part A, we can call upon this route to perform the task by providing it with the region that the user enters. We then wait for a response from the server, and to make sure any errors are caught we use a try catch block to deal with any errors. If no errors occur we will have a list of POIs through which we will iterate through each POI and display them on the HTML page. To do this we will be using the Document Object Model (DOM) which is an underlying mechanism for page manipulation. The DOM offers a whole range of ways to manipulate HTML pages using the concept of nodes, which is essentially an element in a HTML page. For each POI in the list, we create a new node which is basically a new paragraph using a HTML paragraph tag and then we create a new text element that will be used to display the data of the POI. We then append this text to the node as a child, as the DOM and nodes use a hierarchal structure where nodes are at the top and child elements are at the bottom. This node is then appended to a div in the HTML page where the results are to be displayed.

Then I moved on to designing the HTML page that will be used to carry out this task. After giving the page a title, I had to created an input box so the user could enter the region they would like to search for. I gave this input box an ID

and then created a div where the results of the search would be displayed. Last but not least, I created the button that would be clicked to search for the POIs and gave this an ID so the JavaScript page could use an 'onclick' listener and then perform the task described above. Creating this HTML page was a fairly simple process however we still have to connect this HTML page to the JavaScript page we created; to do this we have to insert a script tag in the head that was of type 'module' and had a source of the JavaScript page.

Once I had created this HTML page I linked the HTML elements to the JavaScript from the page earlier and finally I had a working web page.

The fifth task required me to create another HTML page that would be used to add a point of interest and would be linked to the web API created in the second task of part A. This was a simple task. I first created the HTML page that would include a series of input boxes, each with their own IDs, and a submit button that would be linked to the JavaScript page. I then created the JavaScript page and linked it to the HTML page. In this JavaScript page, I created the 'onclick' listener for a new POI being added. This would create a new POI object by creating a "Python Dictionary"/JSON like arrangement of keys and values. Each key would represent the values that would be read in from the HTML page using 'document.getElementById' which takes in the ID of the elements and returns the values. Once I created this object I created a function to add the POI to the database and called this, providing the new POI. This function would perform a HTTP post request by specifying the POI data in the body of the request and then I performed some response checking to determine whether the request was successful or not. I have done this using the status code of the response (refer back to PART A Task 2).

The sixth task in part B was to modify the search results from the region search to include a recommend button for each POI. This allowed me to showcase one of the benefits of using the DOM and nodes as I was able to simply append a button onto the POI node and then use binding to add the 'on click' listener to perform the recommendation by calling a function providing the POI that the user would like to recommend. Binding allows us to attach specific arguments to a call-back which will automatically be received when the function runs. I then created the recommend POI function which simply took the ID of the POI and called the route to perform this task. As this is a new functionality I had to implement the route first. The route would take in the ID using 'req.params.ID' and use this to perform an update SQL query that incremented the

recommendation column by specifying the POI ID. If this was successful the server would return a 200-response code, otherwise it would return a 404 error which informed the user that no POI was found with the specified ID.

PART C

For part C I was required to add simple error checking which is considered good practice when creating web applications. In web application element and this is done using response codes that are sent from the server. These response codes are used by developers to specify a certain outcome whether that be a success or an error. With errors there are many types so there are a variety of response codes.

We were asked to add error checking to the adding to POI functionality (refer to task 2 of part A); to do this we return back to the server file and find the route that carries out the add POI functionality. We must first check that the inputted value isn't blank or null so to do this we can use a pair of 'if' statements that test the values; one will test for null and one will test for blank. If the values are neither null nor blank the server will carry out the task and will return a 200 status code, however if the values are either null or blank a 400 status code will be returned. 200 status codes means that the request was successful and a 400 status code indicates that the server cannot or will not process the request due to a client error which in this case would be a blank or null value. Depending on the status code, the web page will then alert the user whether they successfully added a POI or not.

PART D

For part D we were asked to introduce OpenStreetMap into the application and display the POIs as markers on a map when the user searches for a region and it must also display the name and description of the POI when clicked. To do this, we will make use of the list of POI we created for task 4 to store the POIs retrieved when searching for POIs in a region. However, first we had to initialize the map to display on the screen by creating a div on the HTML page to hold

the map and filling it with a map tile from OpenStreetMap.org. This then loads in the map and we can now begin to add the markers for the POIs. To do this, we must iterate through the list of POIs and first create a location array that holds the longitude and latitude, obtained from the POIs, for the location of the marker and the view location of the map to be specified. We then add the marker to the map. However, as it is the marker is blank as we have not appended any data to it, so to do this we are going to make use of the DOM and node functionality, like we did earlier , we are going to create a new node to represent a new node that will represent the POI to be displayed in the marker pop-up. After creating the node, I also created a text node that uses the POIs information to display the name and description and then appended this to the node, appended the node to the pop-up then binded the pop-up to the marker.

Task 9 then asked us to use the API route created in task 2 to add a new POI to the database at the position on the map that a user clicks on. To do this we must use an 'onClick' listener on the map. When clicked the application will ask the user to enter the POIs details apart from the latitude and longitude as this will be automatically retrieved using the position of the click on the map. A new POI object is created using this information and is sent to a function to add the POI to the database. This function sends a http POST request by providing the POI object as post data and waits for the response from the server. If the response code is 200 then the marker is created at the position of the click and the pop up is added to it. If not a 400 is sent back specifying the error. Using the returned code the function the user is alerted as to whether the POI was recommended or not.

PART E

In this section we were required to implement a session-based login system where the user should be able to login and logout. A message should be displayed on the screen and should remain until the user is logged out even if the page is reloaded.

To implement this, we must first create a session database that will be used to store the session data. After we import the 'express-session' and 'betterSqlite3Session' libraries, we then create a new Sqlite store by using 'betterSqlite3Session' and specifying the 'express-session' and the database.

We then implement the middleware that will create a new session and provide it with the required details such as the Sqlite store we created, a value if the session isn't set and the cookie to name a few.

Now we have created the session functionality, we can move onto creating the server-side code for implementing the login/logout system. First, we will create the login POST route that will be used to login the user. When called using a http POST request, the system will take in the post data which will contain a username and password, and then check the users table to check whether the user exists or not. If they exist, the session variable to represent the username is updated using the username from the post data and a 200 status code is sent back to inform the user that the task was successfully carried out. If not a 401 will be returned to show that there is no user with those credentials.

Now that we have created the login functionality, we can create the logout functionality. To do this, we must create the route that when called upon, simply sets the session to null which will clear the session which essentially means the user is logged out.

As we have been asked to keep the message displayed on the page if the user has logged in even if the page has reloaded, we must also create a route to check this. This login get route will simply return the session username with a null default value.

Now that the server side code has been created for the login and logout functionality, we can implement the front-end using dynamic HTML. Firstly, I created an empty div that will be used to store the either the login form or the 'logged in as …' message that will be filled dynamically.

Moving on the JavaScript page, we must first create and call a function that checks whether the user is logged in, using the login get route. It will check the returned username and if it's null then it will call the 'onLogout' function otherwise it will call the 'onLogin' function providing the username. The 'onLogout' function will display the login form in the div created on the HTML page and will also create a button to submit a login request. An 'onClick' handler is then created for this button and it reads in the username and password and creates a new object that stores the user details and sends it to another 'login' function. This 'login' function sends a http POST request by providing the user details as POST data and if the status code is 200, which means successful, then it will call the 'onLogin' function providing the

username. The ‘onLogin’ function takes in the username and displays a message in the login results div to show the user is logged in. This function is also called if the login check shows the user as logged in using the session username. A logout button is also added to the div and an event handler is also created. When the button is clicked, the logout route is called and the session is cleared. Then the ‘onLogout’ function is called so the login form is displayed again.

Task 11 asked us to use the created login functionality to make sure that the user is logged in to allow them to add a POI. We have also been asked to implement error-checking for this task which means we will have to alter task 9 to display the required error message. To add a POI a http POST request is carried out so to restrict access to this functionality we can simply implement some custom middleware that will only allow access if the user is logged in. Requests such as POST or DELETE may have restricted access as them can change the data so should only be allowed to be accessed by authorised users.

In the server side code, we first create a new function but as opposed to using either ‘app.get’ or ‘app.post’ as we have done before; instead we will use ‘app.use’ which is how middleware is specified as the server runs this code without it having to be called. This route takes both ‘req’&’res’ like normal but also takes a ‘next’ value that will then call the next function. For example, if we wanted to add a new POI we would call the route for it but the middleware would run first and if passed then it will carry out that route. In this middleware, we check whether the request method is either POST or DELETE. If not, the next function is called as it is most likely a GET request which the user doesn’t need to be logged in for. If it is a POST or DELETE request, the middleware will check to see if the user is logged in using the username session variable and if they are they will be allowed to carry out the request but if not a 401 status code is sent back specifying that the user is not logged in. In the front-end we then check for the returned status code and if it’s not 200 then we display the error and let the user know they are not logged in.

PART F

Task 12 asked us to add an additional endpoint to the server to allow users to review a POI. It must first check if there is a POI with the ID specified and the review is not blanked and that the user is logged in.

As mentioned in task 11, I have already created middleware that will only allow a user to perform a POST or DELETE request when they are logged in so this check will be completed automatically when the route is called. Then I created a function route that checked to see whether a POI has the ID specified so once the response is returned if the POI exists then the route will check if the review isn't blank or null and if not it will send the review to the database and a 200 status code will be returned. If any errors occur on the way a 404 error code will be returned with the relevant error attached whether that be no POI exists with the specified ID or the review is blank.

We are then asked to use this route to complete task 13 which required us to add a review box to the pop up we created for task 8. This will use the API we created for task 12 and return an appropriate message to the user. To do this we will use the DOM and node functionality of the pop-up to append two new child nodes to the pop-up. One will be a text node that will be used to input a review and the other will be a button to submit the review. Once these nodes have been appended to the pop-up, we must create the 'onClick' listener that will call the function to carry out the review. This 'onClick' listener will provide the new function with the POI ID, and then in the function we will first retrieve the review using the review input box's ID and create a new review object to represent the review. We will then perform a http POST request which will call the review route we created in task 12 and will wait for the response code. If the user is logged in, the POI exists, and the review isn't blank then a 200 status code will be returned so we can display a success message to the user. If any of these constraints fail, the user will be told that an error has occurred and will be given the context of this error.

```
const revTxt = document.getElementById('revTxt').value
try{
        const poiReview = {
            poi_id : poi_id,
            review : revTxt
        }
```

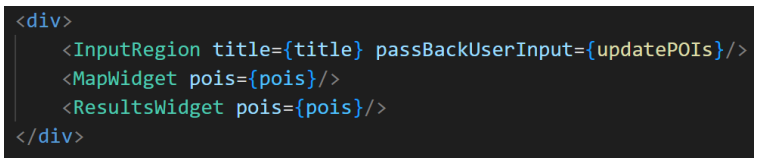
PART G

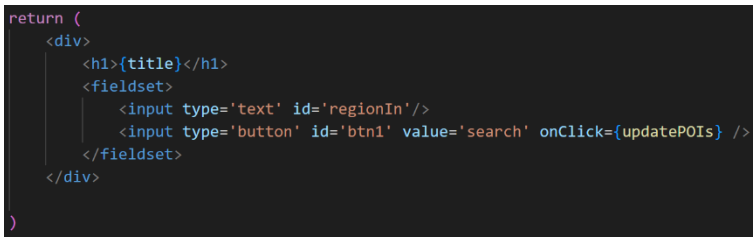
Task 14 asked us to introduce react into the front-end of the web application. React is a JavaScript framework which allows complex JavaScript applications to

be written in easy-to-read, modular code by breaking down the web pages into subsections. To do this we will be using JSX which allows us to embed HTML elements within JavaScript code. This is beneficial when there is a large amount of static HTML and little variable data.

The first step to implement react was breaking down the index page into subsections. The main webpage included 3 main components: the map, the search by region text box and the results div. In a new JavaScript file I created a function that would be used to represent the application and would return three widgets to represent these three components. This 'AppWidget' would hold the state variables which are a store for the applications data and can be accessed when the methods of the 'AppWidget' are called. The state will store the POIs and will also specify the function that will be used to set the POIs. I then created a function to update the POIs which will call upon this 'setPOIs' function given the POIs. We must then specify what will be returned to the HTML page which will be HTML code to display the three components mentioned earlier. We will be using JSX to display the components as mentioned earlier.

The region input box will take in a title and a function to pass back the users input to the 'AppWidget' which is called "lifting state up". This is because in this context the 'AppWidget' acts as a parent of the 3 components so any input must be passed back to the parent so it can be given to the other components. The title and pass back function are known as "Props" which allows us to pass parameters to components, such as the 'InputRegion' widget, by specifying them as attributes in the components tags. This then allows us to embed these properties within JSX. In the 'InputRegion' function we write the HTML code to be returned which will display the input box and the button to submit the search and this button will include an 'onClick' listener which updates the POIs by called another function. This function will retrieve





the region that the user searches for and performs the search using the function we created earlier (see task 4) . When it receives the list of POIs this is then “lifted up” to the ‘AppWidget’ using the pass back function and then the state is updated to store the list of POIs.

The second component is the map which will include all the same functionalities so once we have added these functionalities to the ‘MapWidget’ the map is then returned. The only difference is the map now uses the state POIs to create the markers.

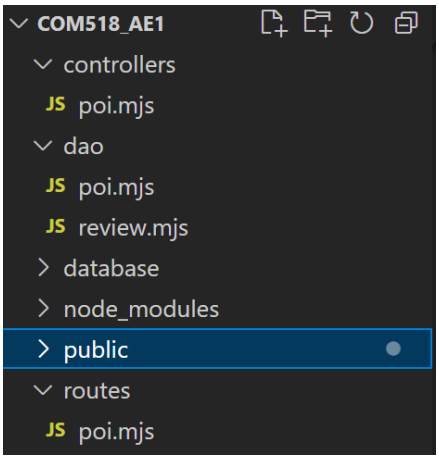
The last component is the ‘ResultsWidget’ which will display the POIs as a list once again using the POIs in the state. This function will simply format the POIs in a user-friendly manner and also append the recommend button to each POI which will call the function that we created in task 6 when clicked. This function will return the formatted POIs as HTML. This is once again making use of JSX as the only the POIs vary in the HTML, the main HTML is static and remains the same as you can see in the image.

```
function ResultsWidget({pois}){
    const poiHtml = pois.map(poi => <li key={poi.id}>   Name: {poi.name}
    return (
        <div>Found POIs:
        <ul>
        {poiHtml}
        </ul>
        </div>
    )
}
```

PART H

The last section of tasks asked us to enhance our web application further to make it more professional. I have used error-checking throughout meaning I can display error or success messaged to the user in a user-friendly manner which makes the website professional. Also the layout of the pages, including having the list of search results below the map, make the site look very professional as well, as the map doesn’t have to move around unnecessarily. Another way to make the web application more professional, is by making the

back-end more well-structuredby using controllers, DAOs, routers and custom middleware (see task 11).



To do this I created all the files that would store all of these components and put them in separate files depending on their type. A router allows us to separate out the routes depending on what they are dealing with. In this example, I have created a POI router that will deal with anything to do with POIs. This will be identified http request by the beginning of the route as if the route starts with '/poi' then this router will be called and then the next part of the route is read and this is then used to select the route that the http request is calling. The router then specifies all the route and calls the required functions from the controller for POIs. We must also bind the controller to the call of the function to preserve the context in callbacks.

```
import db from '../database/db.mjs'
import POIController from '../controllers/poi.mjs';
import session from "express-session";
const pController = new POIController(db)
poiRouter.get('/region/:regionName', pContr
poiRouter.post('/create', pController.creat
poiRouter.post('/recommend/:id', pControlle
poiRouter.get('/check/:id',pController.chec
poiRouter.post('/review',pController.review
export default poiRouter
```

```
class POIDao{
    constructor(db,table){
        this.db = db
        this.table = table
    }
findPOIByRegion(regionIn){
        const stmt = this.db.prepare("SELECT * FROM pointsofinterest WHERE region=?")
        const rows = stmt.all(regionIn)
        if(rows.length == 0){
            return null;
        }
        else{
            return rows
        }
};
```

The controller is used to communicate with the Data Access Object (DAO) to get database results and then format the results so it can be presented to the client. First we must create the DAOs to access the database. As we are interacting with 2 tables in the application ('pointsofinterest' and 'poi_reviews') we will create 2 DAOs one for POIs and one for just reviews. After creating these blank DAOs we can export them in their files and import them into the controller so we can access their functions. We then create a new class to represent the controller that create the DAO objects so they can be accessed. Then we can create the functions that will carry out the tasks required and send them to the intended DAO. These functions will be similar to simple routes as the specify the request

```
class PoiController{
    constructor(db){
        this.dao = new POIDao(db,"pointsofinterest")
        this.dao2 = new ReviewDao(db,"poi_reviews")
    }

findPOIByRegion(req,res){
    try{
        const poi = this.dao.findPOIByRegion(req.params.regionName)
        res.json(poi)
    } catch(error){
        res.status(500).json({error:error})
    }
}
```

and response and also use ‘try-catch’ blocks to implement the task. As opposed to accessing the database straight from here the controller calls the relevant function in the relevant DAO by providing any data that is sent in the request. The controllers functions also handle the response and will set the response code depending on the result of the task. Finally we can move on to designing the DAOs which will deal with all database interactions. The DAOs will take in the database specified in the controller and will use this and the data provided by the controller to interact with the database.

```
class ReviewDao{
    constructor(db,table){
        this.db = db
        this.table = table
    }
reviewPOI(idIn,reviewIn) {
    const stmt = this.db.prepare("INSERT INTO poi_reviews(poi_id,review) VALUES(?,?)")
    const result = stmt.run(idIn,reviewIn)
    return result.changes == 1
}
}
export default ReviewDao
```

To top off the application I decided to add a file-upload facility that would allow a user to upload a photo of any POI. This would be accessible using the marker of the given POI providing the user with an option to select a file to upload an then perform the upload functionality.

To do this, I first implemented the server side code which used a third-party Node module: ‘express-fileupload’. After importing this, I then created the middleware that will be used to check the files that are to be uploaded. This middleware had the role of assigning the files with a temporary directory to be stored in whilst any required checks were being made before uploading. The temporary directory was retrieved from the ‘.env’ file. The ‘.env’ file is linked to the ‘dotenv’ module that allows developers to specify environmental variables such as the temporary directory in this example. After allocating this directory, the middleware also specified a file size limit to make sure that the file that is being uploaded isn’t too large; this limit is also retrieved from the ‘.env’ file. Once we have created this middleware, we can move on to creating the route that will carry out the file-upload. This route takes in the files to upload and first retrieves the filename and then moves the file from the temporary directory to the permanent upload directory which symbolises the file upload

being carried out. This movement of file directory is a promise as it will wait to see if the file can be moved otherwise it is rejected so we must make this functionality 'await' the resolution of the promise and catch any rejection. Due to this, the route must be asynchronous. The permanent upload directory is also specified in the '.env' file. If the file is successfully uploaded, a 200 code will be sent back to show the success, otherwise a 500 code will be send back with the given error.

Now we can implement the front-end to perform the file-upload. To do this, we have to implement a form that will retrieve the photo from the user and submit the photo for upload. This form will have the method of 'post' as it is performing a http POST request and we must also specify the encoding type as 'mulitpart/formdata' because when a file is uploaded it is sent in this format. We must also specify the input type of the file as 'file' which then allows us to select a file to upload and assign an ID. I then added an 'onClick' listener that would call the function to perform the upload. This form was then appended to the marker pop-up for each POI. In the file-upload function, we use the file input's ID to retrieve the files that were selected and first check to see if any files were selected. If not, an error message will be displayed so the user can acknowledge this. If files have been selected, a new 'FormData' object is created which is used to represent POST data which is required for 'multipart/form-data' uploads. The ID of the file input and the file selected are appended to this form data and then the POST request is carried out by specifying the form data in the body of the request. The response from the server is then read and if a 200 is sent back then the user is informed of the success otherwise they are made aware of the error.

To build upon this file-upload facility I have also displayed the image that was uploaded on the markers pop-up for that POI. To do this I created a new div in the pop-up that will hold this image. Then, after retrieving the image from the folder that is storing these images, the image is appended to the pop-up and displayed. The image is then resized using CSS to make it a suitable size for the viewer.

```
if(response.status == 200){
    var src = document.getElementById('photo')
    var img = document.createElement("img")
    img.src = `http://localhost:3030/uploadPics/${POIID+photoFiles[0].name}`
    src.appendChild(img)
    alert("successfully uploaded")
```