A picture containing text

Description automatically generated Solent University Southampton

Faculty of Business, Law and Digital Technologies

Places To Stay Application

COM518

Name: Radoslav Takvoryan

Student number: 14975726

Module title: Web Application Development (COM518)

Module tutor: Nick Whitelegg

Word Count:

Contents

[1. Task 1 3](#_Toc100178118)

[2. Task 2 3](#_Toc100178119)

[3. Task 3 3](#_Toc100178120)

[4. Task 4 3](#_Toc100178121)

[5. Task 5 3](#_Toc100178122)

[6. Task 6 3](#_Toc100178123)

[7. Task 7 3](#_Toc100178124)

[8. Task 8 3](#_Toc100178125)

[9. Task 9 3](#_Toc100178126)

[10. Task 10 3](#_Toc100178127)

[11. Task 11 3](#_Toc100178128)

[12. Part G (Improving my answers) 3](#_Toc100178129)

# Introduction

We need to create an application called Places to Stay. To run the project please unzip the folder and then run in the terminal **npm install ,** after you need to type **node src/main.js** , if you set up the database for the application you should see the server running on port 3000.

# Task 1

The first question is for creating an Express route to find all of the accommodation in a given location in the database. The route is **/location/:location** , it is defined as an Express GET route that is about retrieving data.

locRouter.get('/location/:location', (req, res) => {

db.query(

`SELECT \* FROM accommodation WHERE location = ? `,

[req.params.location],

(error, results, fields) => {

if (error) {

res.status(500).json({ error: error });

} else {

res.json(results);

}

}

);

});

Therefore, we add a GET route to our locRouter which is imported inside our Express application object **main.js**, and provide a callback function that runs whenever a request to that route is made. This callback function takes two parameters, representing the HTTP request and response.

With **query()** method of the connection object, we send a SELECT statement with WHERE condition, the segment inside the URL is to capture the value and populate it inside **req.params.location** /location**/:location**, to the database connection which is **db**, to find all of the accommodations in a given location. Because database querying is asynchronous, we again provide a callback function, which will execute when we receive the results from the database. The first parameter of the callback is an error object. If the error exists, a connection error will have occurred, so we send back an HTTP status code of 500 (Internal Server Error) to the client to indicate that an internal server error occurred, as this error will not be the fault of the user, however server problem, for example, no database running or a programming error in the SQL query. We also send back JSON containing an error message, in this case with the error.

If there is no existing error object, we send back the results, which will be an array of objects, in this case, one accommodation for each row, as JSON to the client.

# Task 2

The second task is about creating an Express route to find accommodation in a given

type and location in the database. The route is /**type/:type/location/:location** with two segments inside the URL, it is defined as an Express GET route which is about retrieving data from the database.

locRouter.get('/type/:type/location/:location', (req, res) => {

db.query(

`SELECT \* FROM accommodation WHERE type=? AND location=?`,

[req.params.type, req.params.location],

(error, results, fields) => {

if (error) {

res.status(500).json({ error: error });

} else {

res.json(results);

}

}

);

});

Therefore, in our locRouter, we add a GET route then import it into our Express application object main.js and provide a callback function to run every time a request is made to the route. The callback function takes two parameters, which are the HTTP request and response.

With **query()** method of the connection object, we send a SELECT with WHERE and AND conditions for two parameters**type** and **location,**the parameters will be populated from the URL segments with the **req.params.type, req.params.location,** to the database connection **db**, to find all of the accommodations within this type and location. The querying is asynchronous due to that we provide a callback function, which will be executed when we receive the response from the database. A first parameter is an error object. If the error exists, a connection error will have occurred, so we send back an HTTP status code of 500 (Internal Server Error) to the client to indicate that an internal server error occurred, as this error will not be the fault of the user, however, it is a server problem, for example, no database running or a programming error in the SQL query. We also send back JSON containing an error message, in this case with the error.

If there is no existing error object, we send back the results, which will be an array of objects, in this case, one accommodation for each row, as JSON to the client.

# Task 3

For task 3 it is a little bit more complicated because we need to create an Express

Route to book accommodation to insert a new row inside the table and to update the existing table inside the database. The route is**/book/:id**with one segment inside the URL which is about the accommodation ID. I found it easier for me to pass it inside the URL. It is defined as an Express POST route which is about adding and modifying data rather than retrieving data.

locRouter.post('/book/:id', (req, res) => {

db.query(

`INSERT INTO acc\_bookings (accID,thedate,npeople) VALUES (?,220601,1) `,

[req.params.id],

(error, results, fields) => {

if (error) {

res.status(500).json({ error: error });

} else {

db.query(

`UPDATE acc\_dates SET availability = availability-1 WHERE accID= ? AND thedate=220601 ; `,

[req.params.id],

(error, results, fields) => {

if (error) {

res.status(500).json({ error: error });

} else {

res.json({ success: results });

}

});

}

});

});

As I have mentioned above in tasks 1 and 2, it is the same process, however, we

are using an Express POST route and it is using two queries rather than one, to execute two tables.

In the first callback, we are checking for an error. If there is an existing error, a connection error will have occurred, so we send back an HTTP status code of 500 (Internal Server Error) to the client to indicate that an internal server error occurred, as this error will not be the fault of the user, however, it is a server problem. We also send back JSON containing the error message with the containing error.

If there is no existing error object, we execute the second query in the same way as the first one with a callback I have mentioned above the first parameter is the error object, if there is no existing error object, we send back the results as a JSON.

# Task 4

Text

Description automatically generated

A screenshot of a computer screen

Description automatically generated with medium confidence

Text

Description automatically generated

For task 4 we need to create an HTML page that allows the user to search for all

accommodation in a given location. Firstly, I have created an HTML file at the top of the file I have specified the DOCTYPE declaration for HTML, it is a piece of information for the browser about what document type to expect. Because I am using Bootstrap I have got a lot of links and scripts inside the head of the HTML, in the bottom of the head I have put my JavaScript which communicates with my REST API, with the type of module to consider the script as a JavaScript module.

Secondly, Inside the body of the HTML, I have divs for the user search input and for the displaying of the results in a user-friendly way which I decide to be inside a table.Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Finally, we have created the JavaScript file inside which we have created an asynchronous function with one parameter location which will be the input from the user. After that, we are using the try-catch statement for the function, if the try block cannot be executed the catch block catch errors. We are using a promise-based function, firstly using fetch API to access the URL (route) from task 1. After that, parsing the results into JSON format. Then looping through the results for each loop which calls each element in an array. Inside it, we are creating the table using the DOM (Document Object Model) elements. we are accessing the table, then creating a body, inside the loop I am creating the elements of the table body (tr, td, th). (**In the provided snips from the code, the code is completed, that is the reason to miss some parts of the code because they are later tasks**). We used another loop for creating the cells of the table which is for a loop. The for loop is created to loop through every key of the accommodation, then again with DOM we create and append new elements. Before the catch block, we use the IF statement to check the length of the table child nodes if they are equal to 2 (2 because there is only a table head and table body) then delete the second child node. This is the way in which we have created the table to reload with the new results every time when the user searches for accommodation in a specific location.

# Task 5

Task 5 was about modifying the code from tasks 3 and 4 and adding BOOK button

inside my AJAX with the REST API (task 3).

Text

Description automatically generated

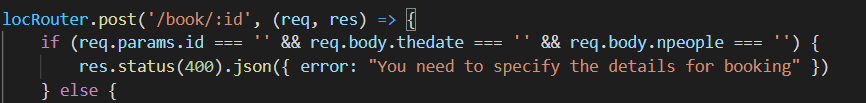
I have created DOM cells for the button for every accommodation to create a button for booking. Then we added an event listener to the button with getting the DOM object (buttonElement). After that, we use fetch to access the route from task 3 and book an accommodation.

# Task 6

For task 6 we need to add error checking to task 3, if any of the details are blank

return an appropriate HTTP code. After that, modify task 5 to test for the HTTP code returned from the server and display user-friendly message to the user.Text

Description automatically generated



I have started firstly by creating inputs for the user to enter the date and the number of people who are inside the bootstrap modal, firstly when you click the book button inside the table the modal will become available and the user can choose the date and the number of people. Then we used the document method with elementID to get the values of the inputs, when we got them we create an empty key-value object where we passed the values of the user with keys for them. Then we fetch the route from task 3 with a POST request because we add and modify data, and headers to be application JSON and in the body passing the object in a JSON string with JSON.stringify. To check the HTTP code of the server we have used the if statement inside which we check if the HTTP code is equal to 400 (Bad Request) then display the red alert to the user with a user-friendly message, otherwise to display the green alert with successfully booked accommodation. In the end, we have added an event listener to the modal close button when you close to reload the window with the location.reload() method which reloads the current URL, it is like the refresh button in the browser.

In the route for task 3, I have added an if statement as well to check if the segment and the values inside the body are empty to send back HTTP code 400 (Bad Request) with a JSON error message containing the message to fill the details.

# Task 7

For task 7 we need to add an OpenStreetMap using Leaflet to task 4 and display

the results as markers on the map when the user clicks on the marker the name and the description should appear as a popup.

Text

Description automatically generated



Text

Description automatically generated



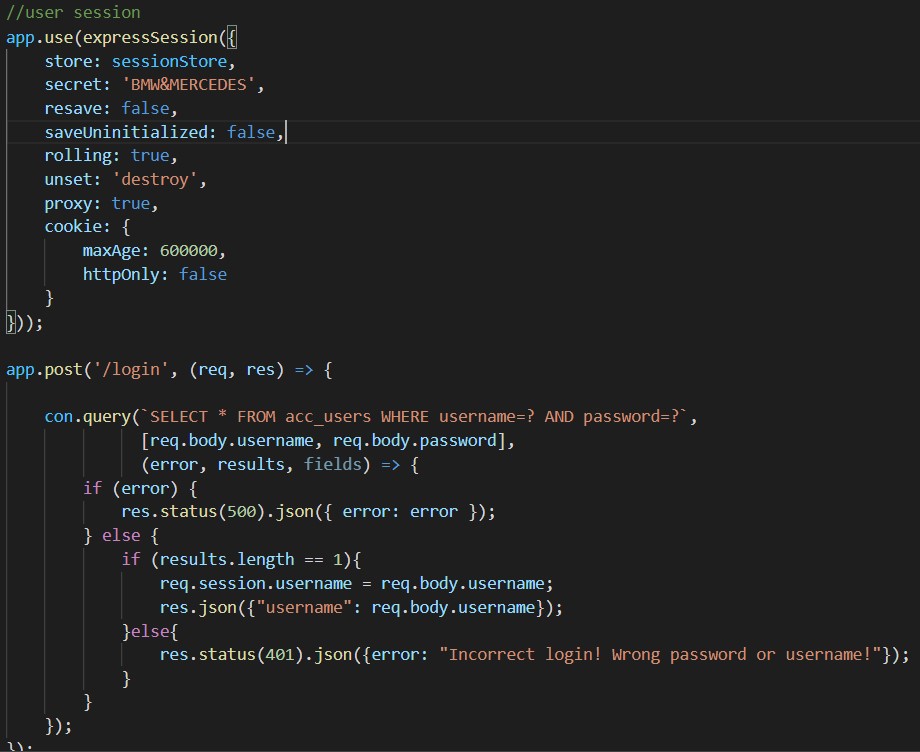
Firstly, inside the HTML page, we link the Leaflet library as an external JavaScript file, as well the Leaflet CSS file because the Leaflet uses some advanced CSS features. Then, inside the HTML file, we are creating a div with an ID of **leafletmap**where the map will be placed.

Secondly, inside my JavaScript file, we firstly create a map object with a variable **map** and an object of **L.Map** created with the **L.map()**method and pass the id of the div inside the brackets (“**leafletmap**”). After that, we set up a map layer because the Leaflet map consists of several layers. The layer, in this case, is **TileLayer**, a layer of map tiles. All of the map tiles come from the server **tile.openstreetmap.org**. The z, x, and y are the z (zoom), x and y are the coordinates. After when we create the layer, we add it to the map **addTo(map)**. We have passed the coordinates inside a two-member array of the latitude and the longitude, in my case, the coordinates are on the Southampton Solent University. In the end, we set up the view of the map to be centered inside the coordinates and the zoom to be 13.

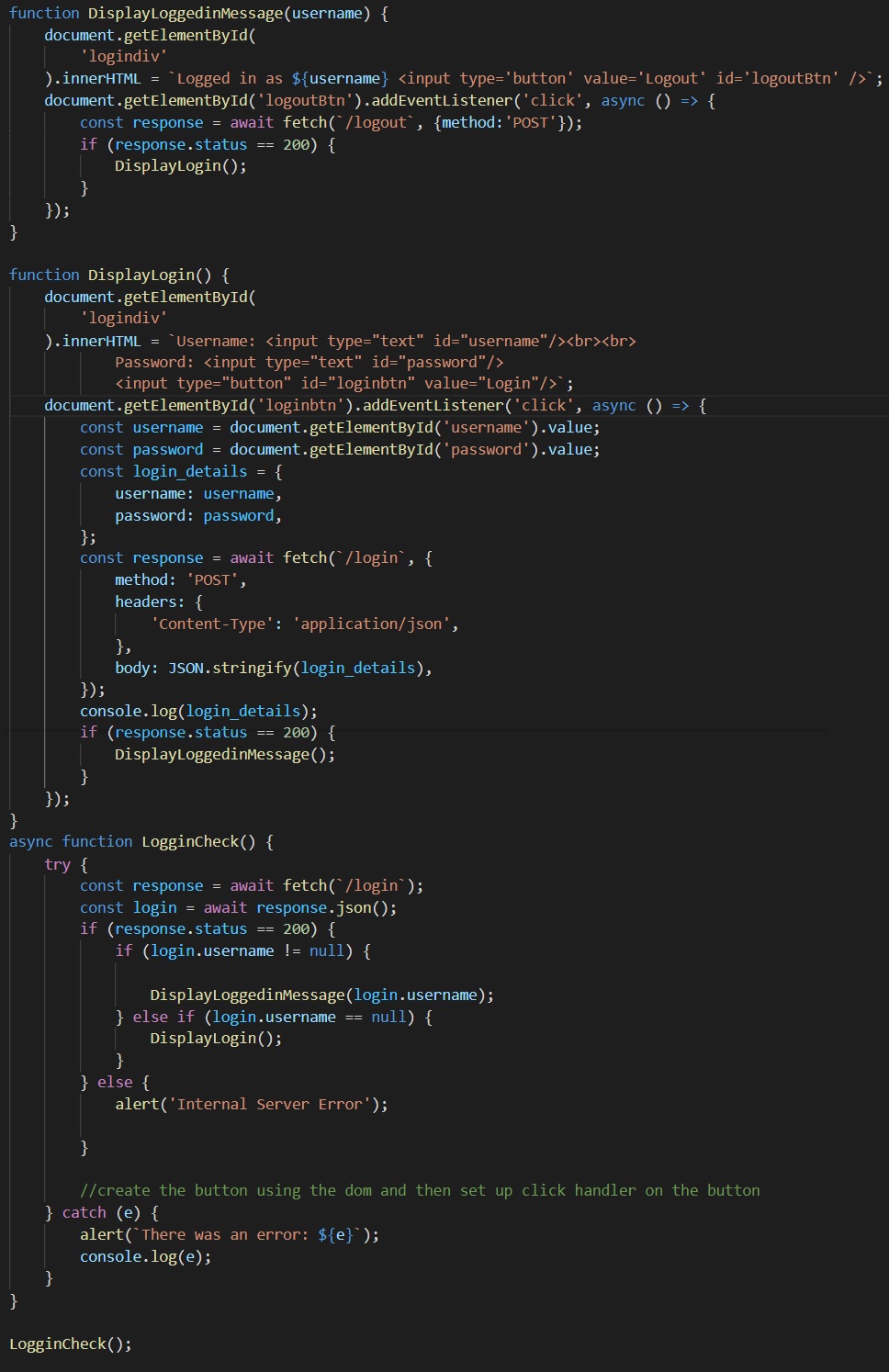
Finally, we are creating the markers with the popups, in the start, we have created one loop to loop through all of the results which are searched by the user. We have declared a variable which is limited inside the **i**which is equal to 0, the other statement is **i to be less than the accommodations.length (the results lenght), and the last one to append one i for every cycle of the loop,**in this case, it will loop through the all keys inside the database for each record. Again we have used the DOM to create paragraphs **p** and then use the createTextNode method to display the name and the description with parameter [i] which is the number of every record (row). Then append them and create a marker at this position with **L.marker** which takes an array of two members, latitude and longitude, attach popup to it with the **bindPopup()** method which takes the div (the paragraphs for name and description) as a parameter and then adds it to the map.

# Task 8

Task 8 is to implement a session-based login system.







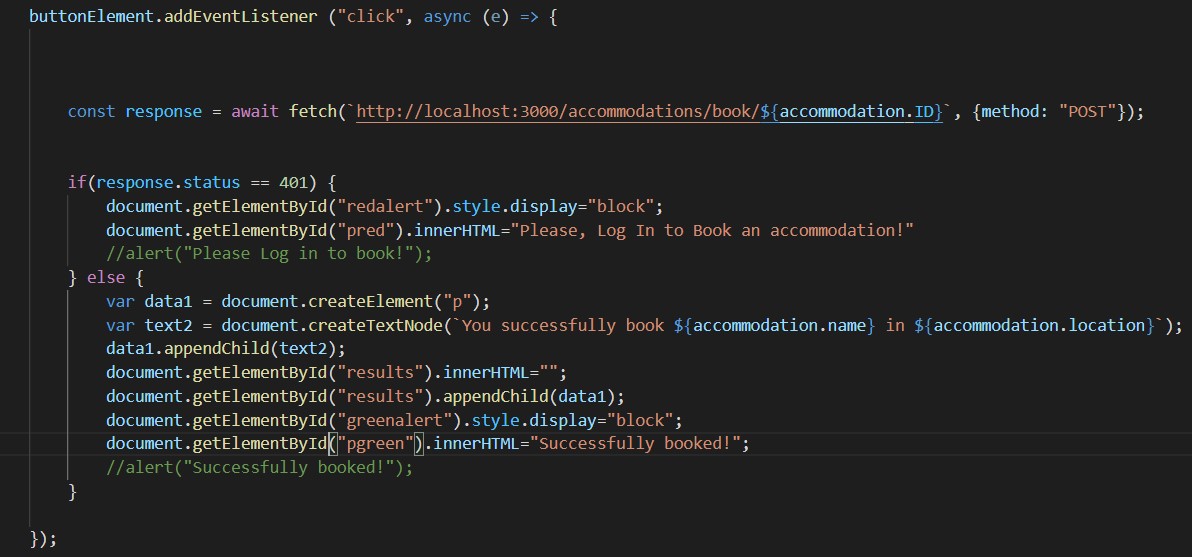
We started with requiring the express-session and the express-mysql-session modules, then we used the connection to create the session store in a promised-based mode. After that we have initialised the Express session object and specified the session store to be used, a secret which is used to digitally sign the session cookie, set up the resave to false, the saveUninitialized, the rolling the unset, the proxy, and the cookie to expire after 10 mins of inactive. Later, we set up the route for getting the details from the user details from the database (there is a similar explanation on it above). However, after checking the error object we check if there are any results and if there are creating the session.username equal to the inserted from the user username. The route for logout is again with a POST request and removing the session. We have got the GET login route which is to request the session username. And in the end, middleware protects any routes using POST or DELETE from access by the user who is not logged in.

Inside the JS file, we have got three functions the first one Is if the user is logged in and the second one is if the user is not logged in. The third one is the important one because it is checking if there is an existing user or there is no one and calling back the appropriate function, in this way it is happening like a loop and it is permanent looping. The third function LogginCheck is again asynchronized function with try-catch block. In the end, am calling back the third function.

# Task 9

Task 9 is about changing tasks 3 and 5 that the user must be logged-in to book

accommodation and if not sending back an appropriate HTTP code. Check the code and send back an appropriate user-friendly message.



After fetching the URL route for booking we check if there is an existing user if there is no the server will return HTTP code 401 (Unauthorised) then we display the red alert with an user-friendly message requesting to log in. Otherwise, displaying green alert for successfully booked accommodation.

# Task 10

Task 10 is about modifying task 3 to check for availability on a specified date.

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

We have created an Express Get route which we can call back after, to use it to check the availability of specific date. It is with two segments inside the URL (the ID of the accommodation and the chosen date from the user).

Inside the JS file I fetch the route with passing the accommodation ID of the chosen accommodation and the chosen date, if the response from the fetching is 200(OK), then check for the availability of the accommodation with getting the availability from the database an compare it with the chosen from user number of people. If there is no availability an appropriate user-friendly message (alert) popup on the window with telling that there is no enough space otherwise process to the booking.

# Task 11

Task 11 is about adding a book button inside the popup and letting the user to enter

the date (user-friendly picking a date) and the number of people.

Text

Description automatically generated

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated

To create the Book button we used the DOM method and then pass it inside the div for the popup, inside the booking we check for any errors, however, the HTTP code 500 is not there because there is any server error the catch block will catch it and again display user-friendly message. We added an event listener to the button DOM element by passing the same code as in tasks 3,7,11. Then I created a loop (in the same way as in task 7) to create an option list for the number of people up to 20 again using the DOM. For the date, we did it by creating DOM elements to display the date in a user-friendly way we have created values and text nodes (the user will only see the text node), however, the date which is sent to the server will be in the same format as it is in the database. To lock the user to not entering different dates from the database we have chosen to use an option list with an option, in this way the user can choose only from them. Lastly, we append everything inside the div to be displayed inside the popup of the marker.

# Part G (Improving my answers)

## Creating Well-structured Node application

For this advanced task, we need to create a well-structured Node application with

DAOs (Data Access Objects), Controllers, Middleware, and Routes.

Firstly, we start with the DAO, with the DAOs we can make our code more structured and object-oriented by passing all of the SQL statements inside the DAOs, It provides an interface to the table as a whole with specified methods. Each method in the DAO performs a particular database operation, for instance, finding all accommodation by a given location. In the end, Each Method returns a Promise Object.

Controllers are part of the MVC (model/view/controller) architecture. The role of the controllers is to communicate with the DAOs to get database results or to get information from the client and insert it into the database.

In routes are all the route files. In the routers, we create controller objects by passing the database connected to them. With the controller object, we handle the routes. We use bind() to save the context of **this** object in the callbacks.

To organise and tidy the project everything is inside sub-folders (controllers, daos, routes, utils) inside the utils are the middleware and the connection to the database.

## Passport for authentication rather than express-session

Text

Description automatically generated

## Text Description automatically generated

For this task, we need to modify the login express-session to Passport. We install the npm packages for the Passport, then require it as a third-party module, import the strategy passport-local, then we link the Passport to our Express application as middleware, the first one **passport-initialize** will initialise the Passport, the second one **passport.session** will set up the Passport for serialisation and deserialization with the session.

We create **/paslogin** route with a POST request, then we call the **Passport authenticate()** method, the strategy must authenticate the user with given credentials, if it fails HTTP code 401 will be sent back. If it is successful the strategy will attach the user details to the request object with **req.user.**Then we use a strategy constructor to create a **LocalStrategy**, we supply it with asynchronous function which takes three parameters (username, password done). The strategy automatically will grab them from the body. Done is a callback function that is necessary to be called to complete the authentication process. Then we check for the username and the password inside the database if they are correct with querying the database. If they are we call back **done()** with null for the first parameter and the user details as a second. Otherwise calling back again **done(),**however, the first parameter is again null and the second one is false which will return HTTP code 401(Unauthorized) or if there is a database error the Passport will generate a 500 (Internal Server Error);

The serialise function saves the user from req.user to the session, the parameter userDetails is the information containing the req.user, however, we save only the ID. For deserialise function we restore the user from the session to req.user, then we perform a database query using the DAO to restore the full user details using the ID, after the full user object is passed to done(), it is with try-catch, if there is any error the catch block will catch it. The logging out is the same for the login system, it will delete the **req.session.passport.user.**To handle authentication errors and supply a custom JSON response to the client we create a separate route **/badlogin**with a GET request to send back the JSON error message. Then we tell to the Passport if the login fails to use the failureRedirect option to **passport.authenticate()**by passing the URL for failed login **/badlogin**.

# Reference

*500+ Hotel Pictures | Download Free Images on Unsplash*a. [viewed Apr 13, 2022]. Available from: <https://unsplash.com/s/photos/hotel>

*Bootstrap Login form - free examples & tutorial*b. [viewed Apr 13, 2022]. Available from: <https://mdbootstrap.com/docs/standard/extended/login/>

CONTRIBUTORS, MARK OTTO, JACOB THORNTON, AND BOOTSTRAP, *Introduction*[viewed Apr 13, 2022]. Available from: <https://getbootstrap.com/docs/4.0/getting-started/introduction/>

Documentation - Leaflet - a JavaScript library for interactive maps. c.

*W3Schools How TO - Code snippets for HTML, CSS and JavaScript*d. [viewed Apr 13, 2022]. Available from: <https://www.w3schools.com/howto/default.asp>