Introduction

This project is a data driven application which uses three Tier Architecture. It means the project has 3 layers.

1. Web Browser: The part of application that user interacts with.
2. Web Server: Like Node.js or Apache
3. Database When the web Browser on user's device send a request to the web server, web browser is a client and web server is a server. On the other hand, when web server sends request to the database, web server is a client and database is a server.

Each database has 2 parts:

1. client side (interactive interface)
2. server side (database back end)

For this project MYSQL which is free and open-source database server and PHPMYADMIN for the database client has been considered. To hosting the database, goormIDE has been used. As we use PHPMYADMIN as a web-based administration tool for working on a MySQL database, any change on the database should be through PHPMYADMIN.

as Node.js can be used in different platforms and it is an open-source server environment, we used Node.js. To processing HTTP requests, Express framework has been used.

To having more modular application and easier-to-maintain, decided to use DAO and Controller structure. There are 3 folders in this project:

1. Routes: It contains only routes and calls related methods.
2. Dao: It contains any infracts with database and any SQL query.
3. Controllers: It handles intraction between routes and daos.

Installing Packages

List of packages has been installed for the project:

1. npm install express
2. node app.js
3. npm install node
4. npm install mysql2
5. npm install cors
6. npm install node-fetch
7. npm install dotenv
8. npm install express-session
9. npm install express-mysql-session
10. npm install passport passport-local

Part A

Question 1 : Look up all accommodation in each location

for this question is needed to interact with accommodation table in the database. so, we need to add 3 files into 3 folders:

1. accomodation-dao.js into Dao folder.
2. accomodation.js into routes folder.
3. accomodation-controller.js into controllers folder.

We start from writing into accomodation.js. This file keeps all routes that are related to accommodation table. First is needed to define a router object for accommodation Routes.

const express = require('express');

const accomodationRouter = express.Router();

Then is needed to require "mysqlconn" to connecting to the database.

const con=require('../mysqlconn');

And Also, require accomodotion-controller.

const AccomodationController = require('../controllers/accomodation-controller');

Now, is needed to Create the controller object, and pass in the database connection as an argument.

const accomoController = new AccomodationController(con);

For this question is needed to have one route which gets location as a parameter and calls related method from accomodotioncontroller. The method would be findaccomodation\_location(We haven't built it yet).

accomodationRouter.get('/location/:location', accomoController.findaccomodation\_location.bind(accomoController));

As can be seen This is a GET request as we just need to get response from the database server, and we don't need to change anything in the database. Also, it calls findaccomodation\_location from controller. bind() is necessary here to preserve the context of this object in call-backs.

The next part would be making accomodation-dao for this route. in DAOs files we have a separate method for each query. We are going to make a findaccomodation\_location method. But first we need to define a class with a constructor.

class AccomodationDao {

constructor(conn, table) {

this.conn = conn;

this.table = table;

}

}

conn is our MySQL database connection and table's name in this case would be accommodation and it will be sent as a parameter from controller. findaccomodation\_location gets location as a parameter and runs the SQL query with passing in the location into SQL query. After running the query is any error happens, err shows the error, if the query returns nothing then it means couldn't find the accommodation and returns null(Tis isn't an error). If none of conditions above happened, means that the result has the correct answer, then it resolves the result.

findaccomodation\_location(location) {

return new Promise ( (resolve, reject) => {

this.conn.query(`SELECT \* FROM ${this.table} WHERE location=?`, [location],

(err, results, fields) => {

if(err) {

reject(err);

} else if (results.length == 0) {

resolve(null);

} else {

resolve(results);

}

});

});

}

at the end of accomodation-dao we need to export the module, then another module can use it.

module.exports = AccomodationDao;

The next step is creating accommodation-controller. Like accomodation-dao it creates a class of AccomodationController with a constructor which Creates a DAO for communicating with the Accommodation table and the database. findaccomodation\_location method calls related method from Dao and sends location as a parameter. This parameter comes from route(/location/:location). After calling Dao method, it returns the result into accommodation variable. then this function sends accommodation as a json file as a response.

class AccomodationController {

constructor(db) {

this.dao = new AccomodationDao(db, 'accommodation');

}

async findaccomodation\_location(req, res) {

try {

const accomodation = await this.dao.findaccomodation\_location(req.params.location);

res.json(accomodation);

} catch (e) {

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

}

}

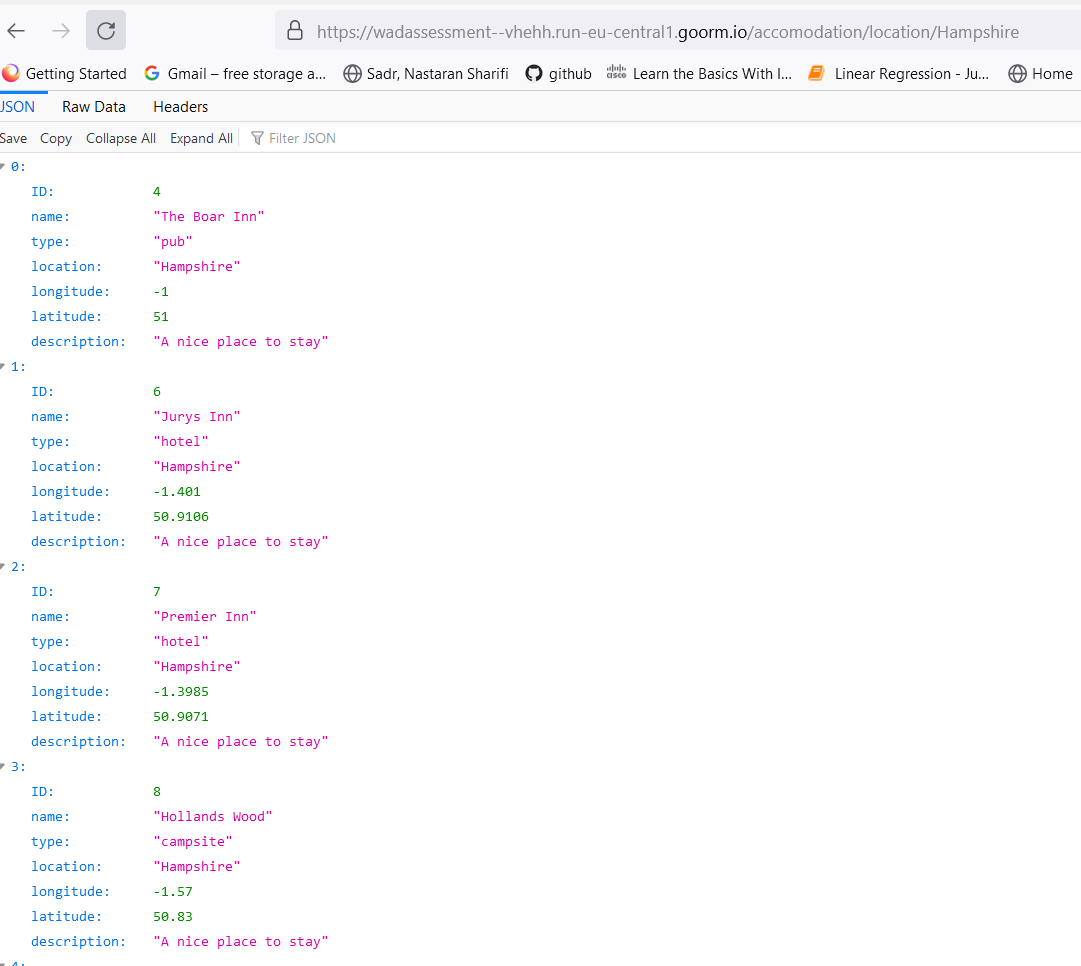
}

module.exports = AccomodationController; // so that the router can use it

Now, we need to tell the app to use accomodationRouter for all routes starts with accommodation. we do it into main.js.

const accomodationRouter = require('./routes/accomodation');

app.use('/accomodation', accomodationRouter);

This part of the application runs without any problem. Test has been done. Picture below shows the test. This example is showing all accommodation in location of "Hampshire".

QUESTION 2: Look up all accommodation of a given type in each location

Like Part A, we are going to add methods to accomodation-dao and accomodation-controller, and also, adding a new router into the accomodation router. The new route would be /location/:location/type/:type. This route gets location and type as parameters to send them to accomoController. Also, it calls findaccomodation\_location\_type method.

accomodationRouter.get('/location/:location/type/:type', accomoController.findaccomodation\_location\_type.bind(accomoController));

findaccomodation\_location\_type method into controller calls related Dao method and sends location and params as parameters.

async findaccomodation\_location\_type(req, res) {

try {

const accomodation = await this.dao.findaccomodation\_location\_type(

req.params.location,

req.params.type

);

res.json(accomodation);

} catch (e) {

console.log(e);

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

}

}

Finally, accomodation-dao would have a new method. the only different with last part is that the query gets 2 parameters.

findaccomodation\_location\_type(location, type) {

return new Promise ( (resolve, reject) => {

this.conn.query(`SELECT \* FROM ${this.table} WHERE location=? AND type=?`, [location, type],

(err, results, fields) => {

if(err) {

reject(err);

} else {

resolve(results);

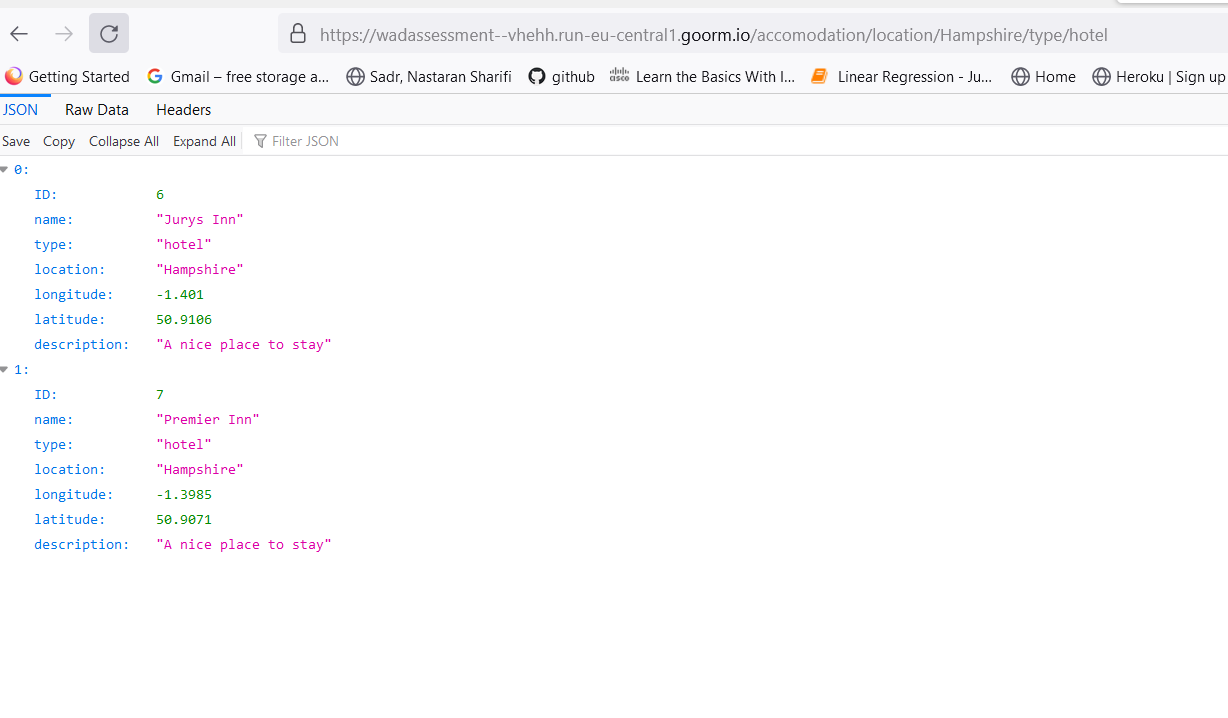
}

});

});

}

This part of the project runs without any problem.



QUESTION 3:book a place of accommodation for a given number of people on a given date and reduce the availability in the acc\_dates table.

for this question the APP is going to interact with acc-bookings and acc-dates table. This task includes a few steps:

1. Create a new route for booking a place. As this part is related to acc-bookings table, we are going to add new files in route/controller/dao folder for acc-bookings.
2. the next step is reducing availability in acc-dates table. For this part is needed to add new files in route/controller/dao folder for acc-dates. We wouldn't use this route directly. Only acc-booking controller would call it to reduce the availability after booking.

new route for this part in acc-bookings.js is"/book". The reason that it doesn't get any parameter is that all needed parameters will be come through body of HTTP request. It also, calls book method for controller. Type of request would be post as tables would be changed.

acc\_bookingsRouter.post('/book', accbController.book.bind(accbController));

acc-bookings-dao.js includes the query that add a row in acc-bookings table. It gets accid,date and number of people that come from controller. In this query insert will be used.

book(accid,date,number) {

return new Promise ( (resolve, reject) => {

*//cosole.log(accid)*

this.conn.query(`INSERT INTO ${this.table}(accID,thedate,npeople) VALUES (?,?,?)`, [accid,date,number],

(err, results, fields) => {

if(err) {

*//cosole.log(err)*

reject(err);

} else {

resolve(results.insertId); *// resolve with the record's allocated ID*

}

});

});

}

The controller for this part calls 2 daos. So, into constructor there are 2 daos.

constructor(db) {

*// Create DAO for communicating with the databases*

this.dao = new Acc\_bookingsDao(db, "acc\_bookings");

this.datesdao=new Acc\_datesDao(db,"acc\_dates")

}

As has been mentioned above the first part class book method in acc-booking-dao to inserting a new row for the booking. It gets all needed parameters from the body of request. the second part calls reduce\_avalibility method from acc-dates-dao to reducing the availability in acc-dates table. It only gets accid as a parameter.

async book(req, res) {

try {

*//cosole.log("controller")*

*//console.log(req.body.accid)*

const accid = await this.dao.book(req.body.accID, req.body.thedate,req.body.npeople);

const acc\_d = await this.datesdao.reduce\_avalibility(accid);

*//console.log(accid)*

res.json({id: accid});

} catch(e) {

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

}

}

As we have a new table, is needed to add related controller/dao/routes for to handle all the income request to this table. acc-date.js has been added for the route. This is a post request, and it gets accid as a parameter.

acc\_datesRouter.post('/reduce/:accid', accdController.reduce\_avalibility.bind(accdController));

acc-dates-dao is same as other daos but only the query will be changed to an update query.

reduce\_avalibility(accid) {

return new Promise ( (resolve, reject) => {

this.conn.query(`UPDATE ${this.table} SET availability=availability-1 WHERE accID=?`, [accid],

(err, results, fields) => {

if(err) {

reject(err);

} else {

resolve(results.insertId); *// resolve with the record's allocated ID*

}

});

});

}

And also, the controller will be added to acc-dates-controller.js:

async reduce\_avalibility(req, res) {

try {

const accid = await this.dao.reduce\_avalibility(req.params.accid);

res.json({id: accid});

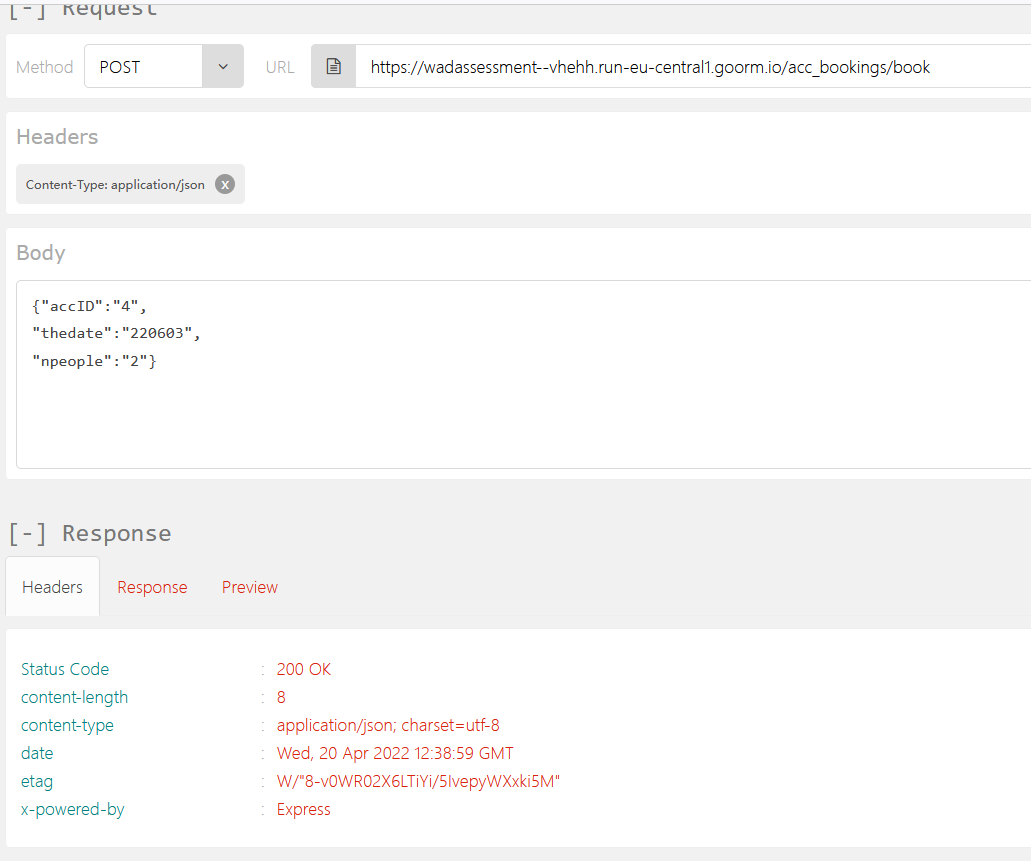
} catch(e) {

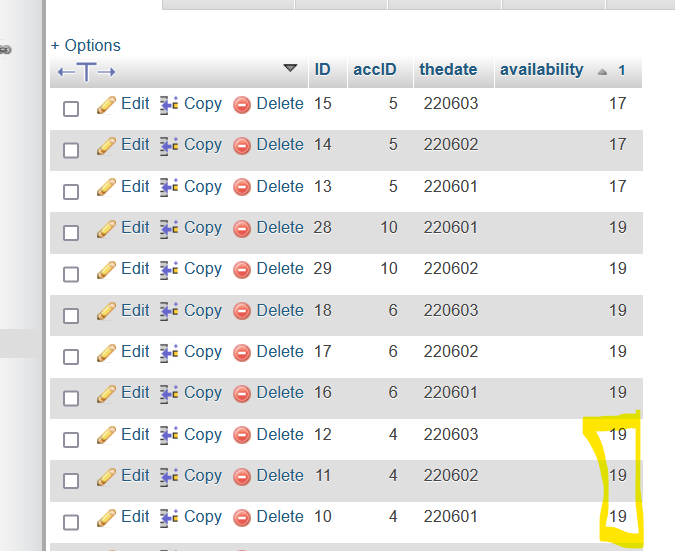
res.status(500).json({error: e});

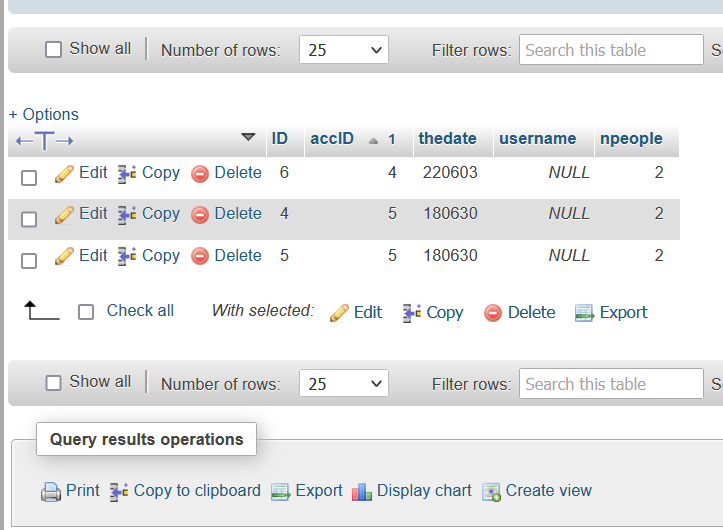
}

}

This part of code runs without any problem. As this is a post request is needed to use Rester to test it.







Part B

Question 4: Write an HTML page which allows the user to search for all accommodation in each location.

For this task we need to add a new folder "public". In this folder we need to add 2 files:

1. index.HTML: include all HTML code to that the user can see.
2. index.js: include all client code which runs in user's browser. Next is needed to add a textbox and a button in HTML file to let the user add the Location and also, press the button to get the result.

Location: <input id="searchterm" />

<input type="button" id="ajaxButton" value="Search!" />

<div id="userdetail"></div>

<div id="results"></div>

The next step is completing the client-side code. first is needed to add an event listener on the button we defined in HTML. Then the event listener be called when the user presses the button. With pressing the button by user, an AJAX post request will be sent to the Rest API we have added already to the server-side code. It is needed to send a post request to "accomodation//location/:location". "location" parameter into the route URL will come from the textbox in HTML file. The code below is the event listener that calls the function for handling POST request.

*// Make the AJAX run when we click a button*

document.getElementById('ajaxButton').addEventListener('click', () => {

*// Read the location from a text field*

console.log("ajaxstarted")

const searchterm = document.getElementById('searchterm').value;

ajaxSearch(searchterm);

});

When the event listener runs, It gets the search term which is location and send it to "ajaxSearch" function."ajaxSearch" function, first fetch the API, Then Pars the jason.

const response = await fetch(`/accomodation/location/${searchterm}`);

const accomodations = await response.json();

After parsing json all information about accomodation is in accommodations array. That information are "name,type,location,longitude,latitude and description". So, to accessing that information is needed to have a loop into the array. before that is needed to have an HTML table to place information. The code bellow shows how to embed HTML code into Java:

let html = `<table>`;

html += `<tr>`;

html += `<th> Name</th>`;

html += `<th> Type</th>`;

html += `<th> Location</th>`;

html += `<th> Description </th>`;

html += `</tr>`;

html += `</table>`;

The code below is looping through the array. For each item of the array, it creates a new paragraph and then creates a text variable. Into the Text variable embeds details of accomodation. To do that is needed to embed the field's name into ${} and using ``. Then is needed to add the Text variable into HTML page, to doing that we use appendchild(text1) to the paragraph we already added. The last step is adding to node1 to the results which is an HTML file.

accomodations.forEach((accomodation) => {

*/\* Create a brand new paragraph element \*/*

var node1 = document.createElement('p');

*/\* Create a brand new text node \*/*

var text1 = document.createTextNode(

`${accomodation.name} -- ${accomodation.type} --${accomodation.location} --${accomodation.description}`

);

*/\* Add the text node to the paragraph \*/*

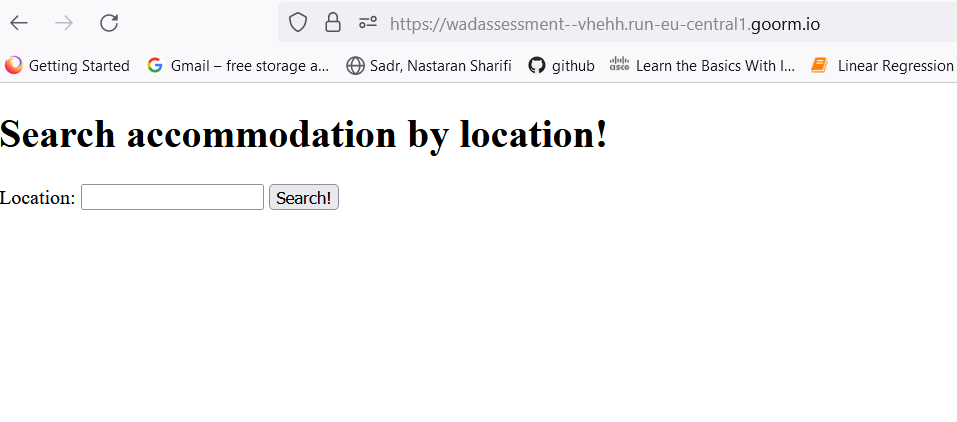
node1.appendChild(text1);

*/\* Add the new node to the results <div> \*/*

document.getElementById('results').appendChild(node1);

});

This part of the Project runs without any problem.



Question 5: Modify the search results, so that you create a “Book” button for each result.

For this task is needed to add a book button for each result. To doing that, we need to appendchild a button inside the loop.

var btn = document.createElement('input');

document.getElementById('results').appendChild(btn);

btn.setAttribute('type', 'button');

btn.setAttribute('value', 'Book');

Then defending another event listener on the new button. The event listener will Fetch API "/acc\_bookings/book". Remember /acc\_bookings/book is a Post request which the body of the request includes all needed item for the Post request. So, before adding the event listener we are needed to add a jason file include accomodation id, thedate of booking and number of people. At this point number of people is 1 and thedate would be one of those date into acc-dates table for example, 220601.

const accID = accomodation.ID;

const thedate="220601";

const npeople= "1";

const acc = {

"accID": accID,

"thedate": thedate,

"npeople": npeople

};

When book button presses by user, Fetch API instruction will run Post request with the body of json we already defined. Then if response of the Post request is 200, means everything is fine, otherwise, an error alert will be shown to the user.

btn.addEventListener('click', async (e) => {

console.log("event listener")

const response = await fetch(`/acc\_bookings/book`, {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify(acc),

});

if (response.status == 200) {

var text2=`Accomodation with "ID:${accomodation.ID}" "Name:${accomodation.name}" Successfully Booked!`

alert(text2);

}else {

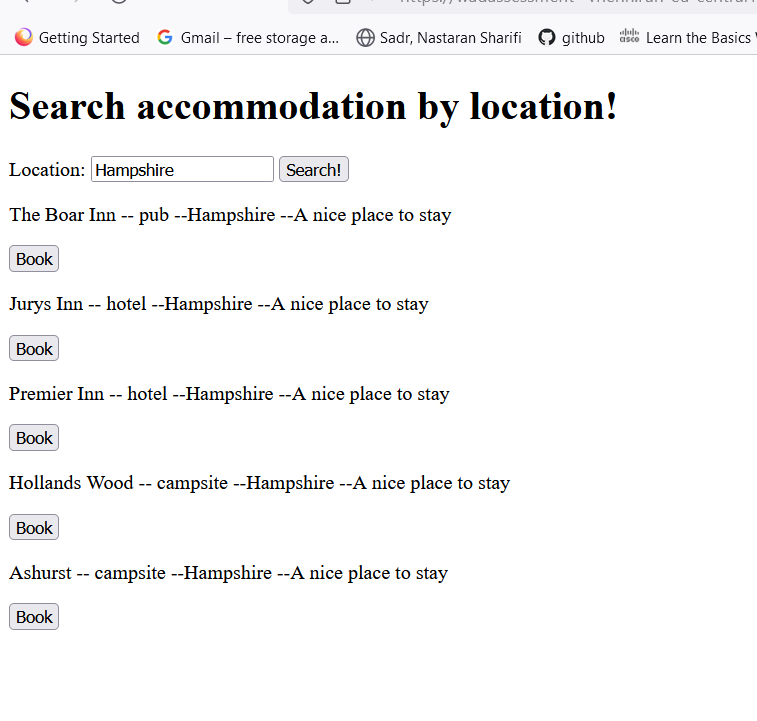
alert('Something went wrong!');

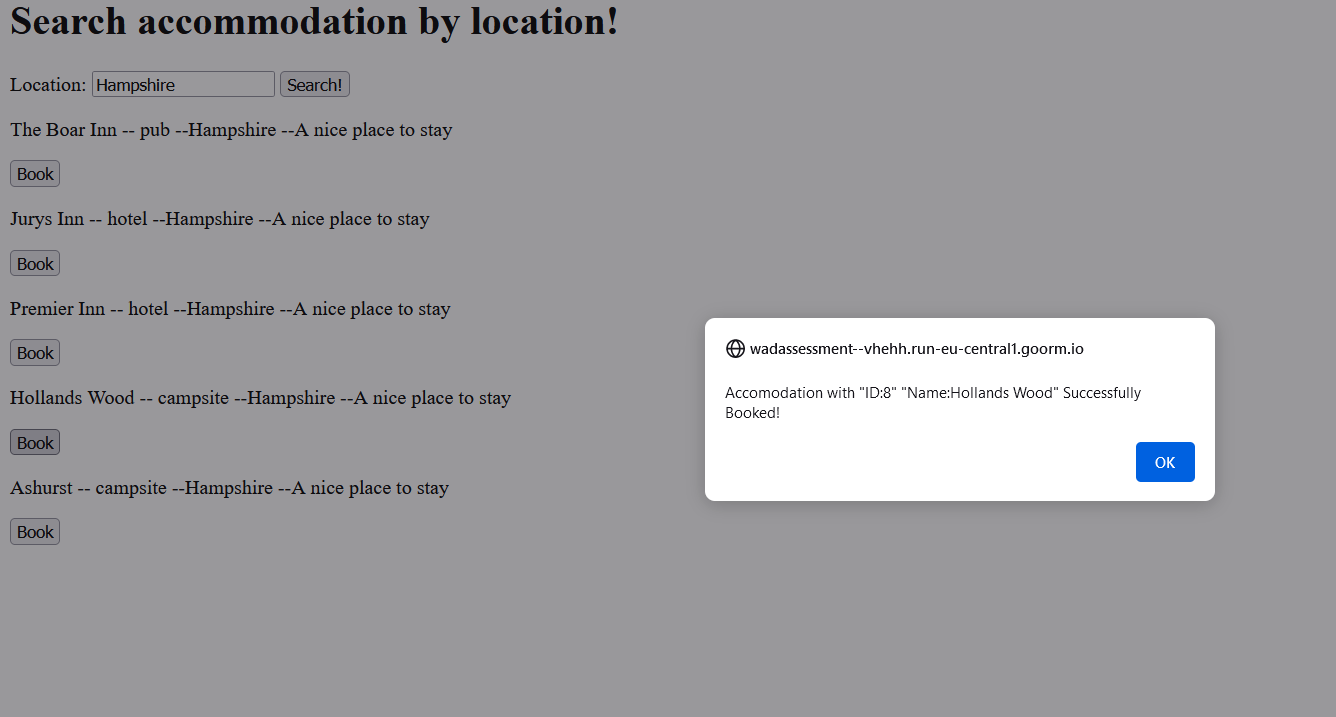
}

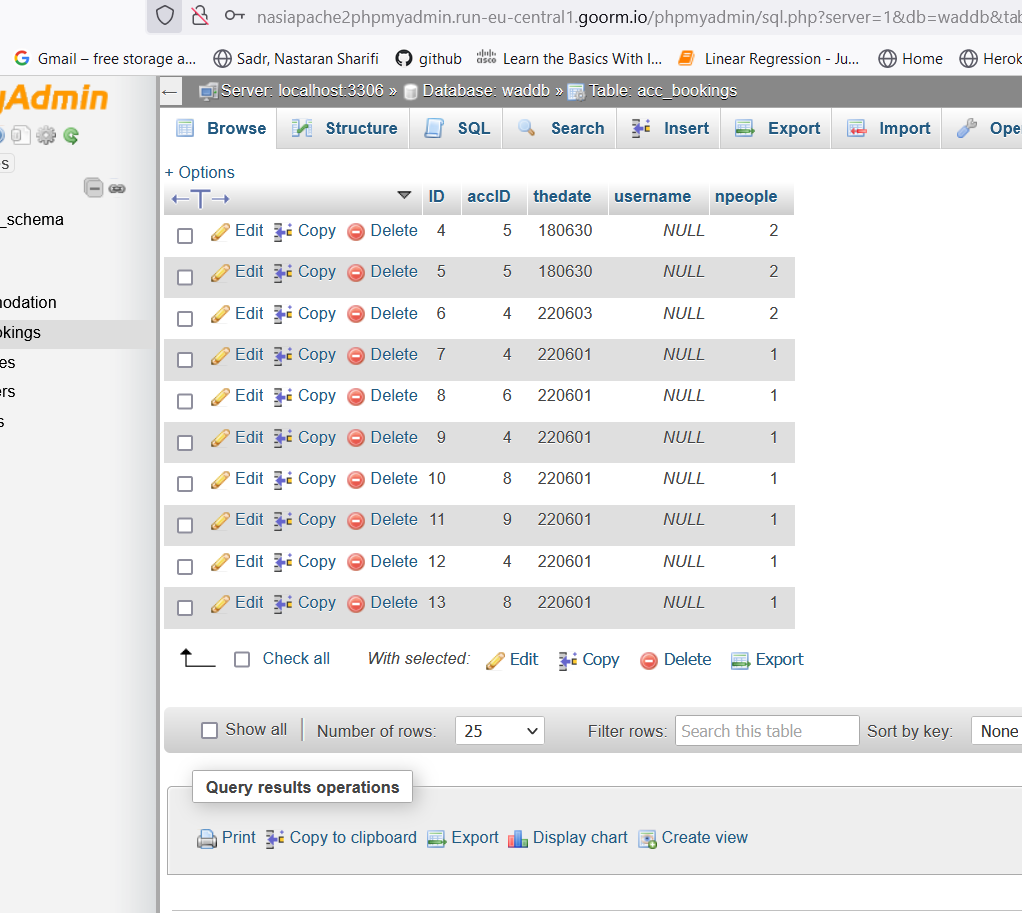
});

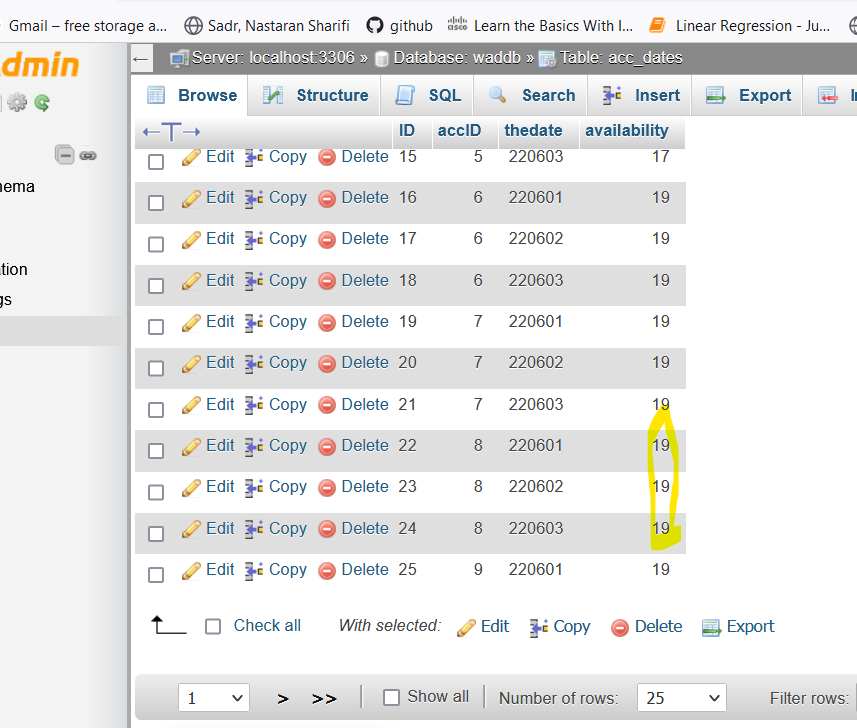
});

This part of code executes without any problem.









Part C

Adding simple error-checking

To Protecting booking route to showing an error message if one of those files(npeople, accID, thedate) are null, is needed to add error checker before getting any server error. Server error will happen when we send null field through SQL query. As we do not want to have a server error, is needed to control it before access the database. It means acc-bookings-controller must handle this error and if one of those field are empty, it sends 401 error response which is a client-side error. In the code below

try {

if (req.body.accID && req.body.thedate && req.body.npeople){

const accid = await this.dao.book(req.body.accID, req.body.thedate,req.body.npeople);

const acc\_d = await this.datesdao.reduce\_avalibility(accid);

res.json({id: accid});

}else{

res.status(401).json({error: "Bad request"});

}

} catch(e) {

console.log(e)

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

}

On the other hand, is needed to check if the response is 401, a message will be shown to the user.

if (response.status == 200) {

var text2=`Accomodation with "ID:${accomodation.ID}" "Name:${accomodation.name}" Successfully Booked!`

alert(text2);

}else if (response.status==401) {

alert('Please fill all the fields!');

}else if (response.status==500) {

alert('server error');

}

This part runs without any problem. To testing this part by purpose one of the files in jason changed to null.

const accID = accomodation.ID;

const thedate="220601";

const npeople= "1";

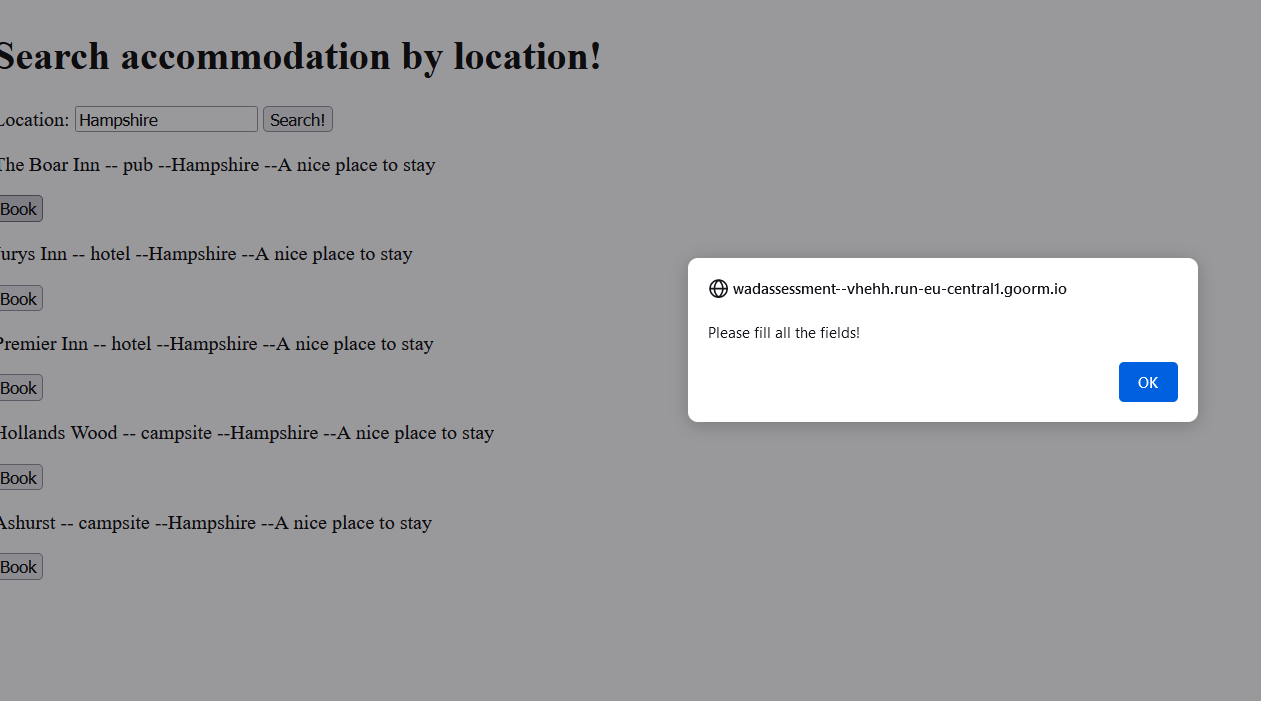
const acc = {

"accID": accID,

"thedate": thedate,

"npeople": npeople

};



Part D

Adding Map

First, We link in the Leaflet library as an external JavaScript file and also, link in the Leaflet CSS file.

<link rel="stylesheet" href="https://unpkg.com/leaflet@1.7.1/dist/leaflet.css" />

<script type='text/javascript' src='https://unpkg.com/leaflet@1.7.1/dist/leaflet.js'></script>

Then add a map div with ID of map1 in HTML.

<div id="map1" style="width: 800px; height: 600px;"></div>

<br />

Now is needed to change ajaxsearch function so that instead of showing a table as a result, shows the result on the map. After Fetching "/accomodation/location" and parsing json into accommodations array, is needed to set up a map layer. I decided to set the first view of map in latitude:51 longitude:-1 and zoomlevel:14.

const attrib="Map data copyright OpenStreetMap contributors, Open Database Licence";

*//set up a map layer*

L.tileLayer

("https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png",

{ attribution: attrib } ).addTo(map);

*//set the latitude and longitude of the centre of the map, here latitude=50.908,*

*//longitude=-1.4. We pass the latitude and longitude in as a two-member array.*

map.setView([51,-1], 14);

Like part B is needed to loop through each item in array. Into the loop, we first define a set of latitude and longitude of the array's item into "pos". then we add a marker in the location of pos in the map. Also, a bindPoppup added to the map, then when the user clicks to the any marker, it shows the information of that accommodation. Then with using an event handler on the map, when the user clicks in any area of the map, it shows latitude and longitude of that position.

accomodations.forEach((accomodation) => {

const pos = [accomodation.latitude, accomodation.longitude];

L.marker(pos).addTo(map);

var text2=` "Name:${accomodation.name}" "Description:${accomodation.description}"`

const marker = L.marker(pos).addTo(map);

marker.bindPopup(text2);

map.on("click", e => {

*// "e.latlng" is an object (of type L.LatLng) representing the mouse click*

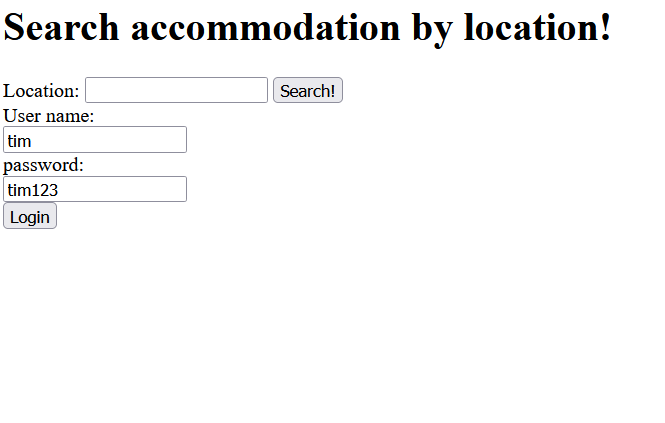
*// position*

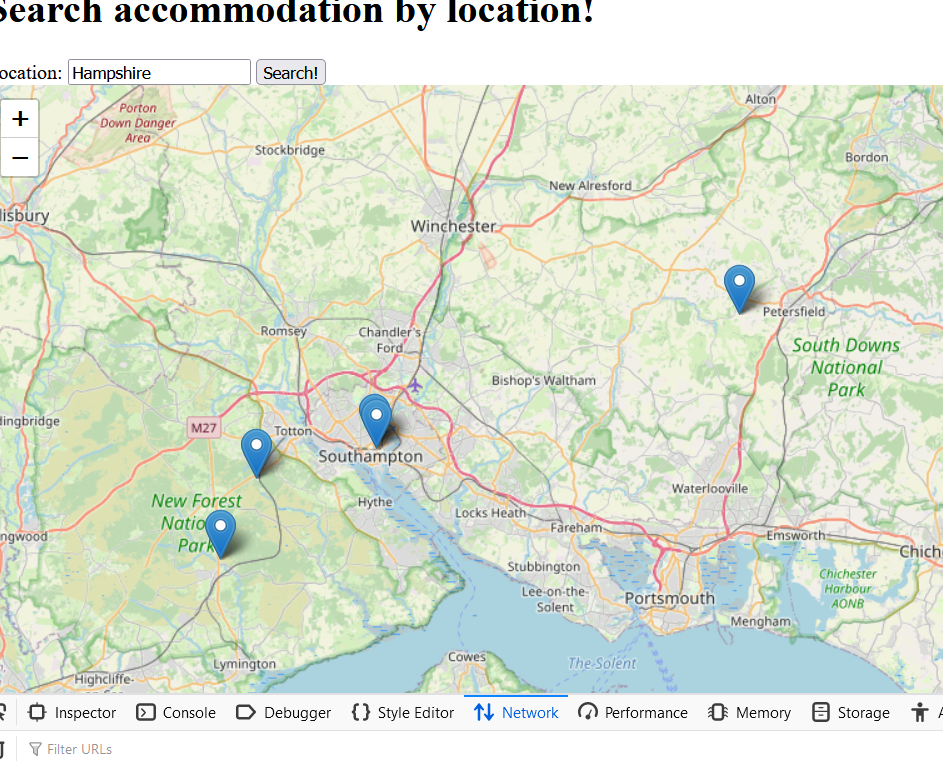
*// It has two properties, "lat" is the latitude and "lng" is the longitude.*

alert(`You clicked at:${e.latlng.lat} ${e.latlng.lng}`);

});

});

This part of the code runs without any problem.

Part E

Question8:Implement a session-based login system

To implementing login a session-based login system first is needed to implement all related routes. For this purpose, 3 routes is needed:

1. Post login route
2. Get login route: useful for clients to obtain currently logged in user
3. Post logout

So, authontication.js in routes folder includes all the routes, authontication-dao.js includes all daos and authontication-controller.js includes all controllers.

The first step is adding 3 routes:

AuthonicationRouter.post('/login', sController.userlogin.bind(sController));

AuthonicationRouter.post('/logout', sController.userlogout.bind(sController));

AuthonicationRouter.get('/login', sController.usergetlogin.bind(sController));

The Next step is adding DAO. The only route that needs Dao is POST LOGIN route as GET LOGIN and POST LOGOUT don't need any query from the database. After running SQL query if the result's length is 0 then dao returns null otherwise it returns the results.

userlogin(username,password) {

return new Promise ( (resolve, reject) => {

this.conn.query(`SELECT \* FROM ${this.table} WHERE username=? AND password=?`,[username,password],

(err, results, fields) => {

if(err) {

console.log(err)

reject(err);

} else if (results.length == 0) {

*// resolve with null if no results - this is not considered an error, so we do not reject*

resolve(null);

} else {

*// only one student will be found but "results" will still be an array with one member.*

*// To simplify code which makes use of the DAO, extract the one and only row from the array*

*// and resolve with that.*

resolve(results);

}

});

});

}

}

In controller, for post login route is needed to call the userlogin method in authentication-dao. If the result of userlogin method which goes to user variable , is null then it means username or password is wrong and returns 404 error message otherwise it changes the session's username to the user's username and as a response it returns user's username into a 200 HTTP response message.

sync userlogin(req, res) {

try {

console.log("controller")

console.log(req.body.username)

console.log(req.body.password)

const user = await this.dao.userlogin(req.body.username,req.body.password);

*// Remember from the DAO that the promise resolves with null if there are no result*

console.log(user)

if(user == null) {

res.status(404).json({error: "username or password is wrong"});

} else {

req.session.username = req.body.username;

res.status(200).json({username:req.body.username})

}

} catch(e) {

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

}

}

Two more methods are needed to be add in the controller. Get login and Post logout. In userlogout method, req.session changes to null and sends a json message. And in Usergetlogin method checks req.session.username. If it is null means user hasn’t logged in otherwise means the user already logged in.

async userlogout(req, res) {

try {

req.session = null;

res.json({'success': 1 });

} catch(e) {

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

}

}

async usergetlogin(req, res) {

try {

res.json({username: req.session.username || null} );

} catch(e) {

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

}

}

Then in setup.js is needed to setup a session module. first, we require expressSession and express-Mysql-session modules.

const expressSession = require('express-session');

const MySQLStore = require('express-mysql-session')(expressSession);

Now is needed to specify quite a few options for the session in setup.js:

app.use(expressSession({

*// Specify the session store to be used.*

store: sessionStore,

*// a secret used to digitally sign session cookie, use something unguessable (e.g. random bytes as hex) in a real application.*

secret: 'BinnieAndClyde',

*// use as recommended by your chosen session store - related to internals of how session stores work*

resave: false,

*// save session to store before modification*

saveUninitialized: false,

*// reset cookie for every HTTP response. The cookie expiration time will be reset, to 'maxAge' milliseconds beyond the time of the response.*

*// Thus, the session cookie will expire after 10 mins of \*inactivity\* (no HTTP request made and consequently no response sent) when 'rolling' is true.*

*// If 'rolling' is false, the session cookie would expire after 10 minutes even if the user was interacting with the site, which would be very*

*// annoying - so true is the sensible setting.*

rolling: true,

*// destroy session (remove it from the data store) when it is set to null, deleted etc*

unset: 'destroy',

*// useful if using a proxy to access your server, as you will probably be doing in a production environment: this allows the session cookie to pass through the proxy*

proxy: true,

*// properties of session cookie*

cookie: {

maxAge: 600000, *// 600000 ms = 10 mins expiry time*

httpOnly: false *// allow client-side code to access the cookie, otherwise it's kept to the HTTP messages*

}

}));

The next step is adding HTML login form in index.HTML.

<div id="loginform">

<label for="username" id="usernamelbl">User name:</label>

<input type="text" id="username" name="username">

<label for="password" id="passwordlbl">password:</label>

<input type="text" id="password" name="password">

<button type="button" id="ajaxloginbtn" value="login!">Login</button>

<button type="button" id="btnlogout" value="logout!">Logout</button>

</div>

Also, is needed to change index.js. In index.js, loadpage\_event() function added which runs when the browser loads. This function by using Get login route, checks if the user already logged in or not. If the user hasn't logged in, then createlogginform() function will be called otherwise createlogginmessage() function will be called.

async function loadpage\_event() {

try {

const response = await fetch(`/authontication/login`);

const user = await response.json();

console.log("aaaaa")

console.log(user)

if (user.username == null) {

createlogginform();

} else {

createlogginmessage(user);

console.log(user.username)

}

} catch (e) {

alert(`There was an error: ${e}`);

}

}

loadpage\_event();

In createlogginform() function, first makes all login form's elements visible:

document.getElementById('username').style.display = 'block';

document.getElementById('password').style.display = 'block';

document.getElementById('ajaxloginbtn').style.display = 'block';

document.getElementById('passwordlbl').style.display = 'block';

document.getElementById('usernamelbl').style.display = 'block';

document.getElementById('btnlogout').style.display = 'none';

document.getElementById('userdetail').style.display = 'none';

Then create an event listener on the login form's button. The event listener first reads username and password entered by user, then creates a user object. Then fetch login route. It is important to send the json object in the body of post request. Then checks the response back from the http post request. If it is 200 means login done successfully and it calls createlogginmessage() function. Otherwise, it shows an error message.

ajaxloginbtn.addEventListener('click', async (e) => {

*//getting username and passport*

const username = document.getElementById('username').value;

const password = document.getElementById('password').value;

*//creating a user object*

const user = {

username: username,

password: password,

};

console.log("bbbbb")

console.log(user)

const response = await fetch(`/authontication/login`, {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify(user),

});

if (response.status == 200) {

*// Parse the JSON.*

*// const user2 = await response.json();*

console.log("login route successful");

createlogginmessage(user);

} else {

alert('Username or Password is wrong');

}

});

Createlogginmessage(), first changes the visibility of form items to none. But it makes the btnlogout and userdetail visible. Userdetail is a label to show the login message to the user.

document.getElementById('username').style.display = 'none';

document.getElementById('password').style.display = 'none';

document.getElementById('ajaxloginbtn').style.display = 'none';

document.getElementById('passwordlbl').style.display = 'none';

document.getElementById('usernamelbl').style.display = 'none';

document.getElementById('btnlogout').style.display = 'block';

document.getElementById('userdetail').style.display = 'block';

Then creates a login message for the user. After login the user needs to have a logout button too. Then it fetches logout route and checks the response back to showing a message to the user.

document.getElementById('userdetail').innerHTML=`Logged in as ${user.username}`

*/\* Add the new node to the results <div> \*/*

*//document.getElementById('results').appendChild(node1);*

btnlogout.addEventListener('click', async (e) => {

const response = await fetch(`authontication/logout`, {

method: 'POST',

});

if (response.status == 200) {

alert('Successfully Logged out');

*//calling loggin form*

createlogginform();

} else if (response.status == 401) {

alert('you are not logged in');

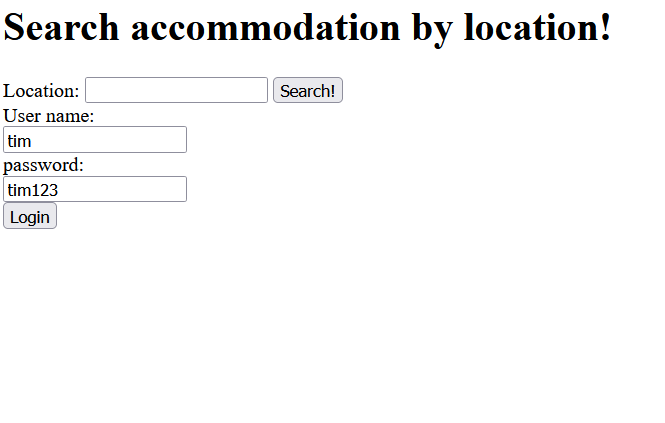
} else {

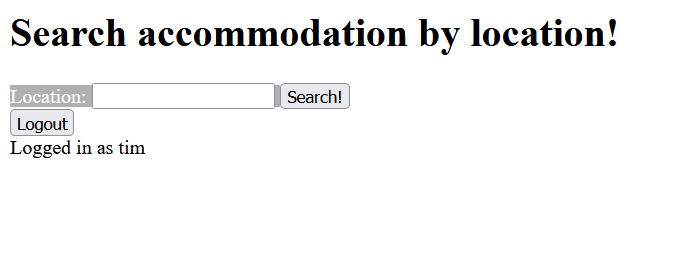
alert('There is somthing wrong');

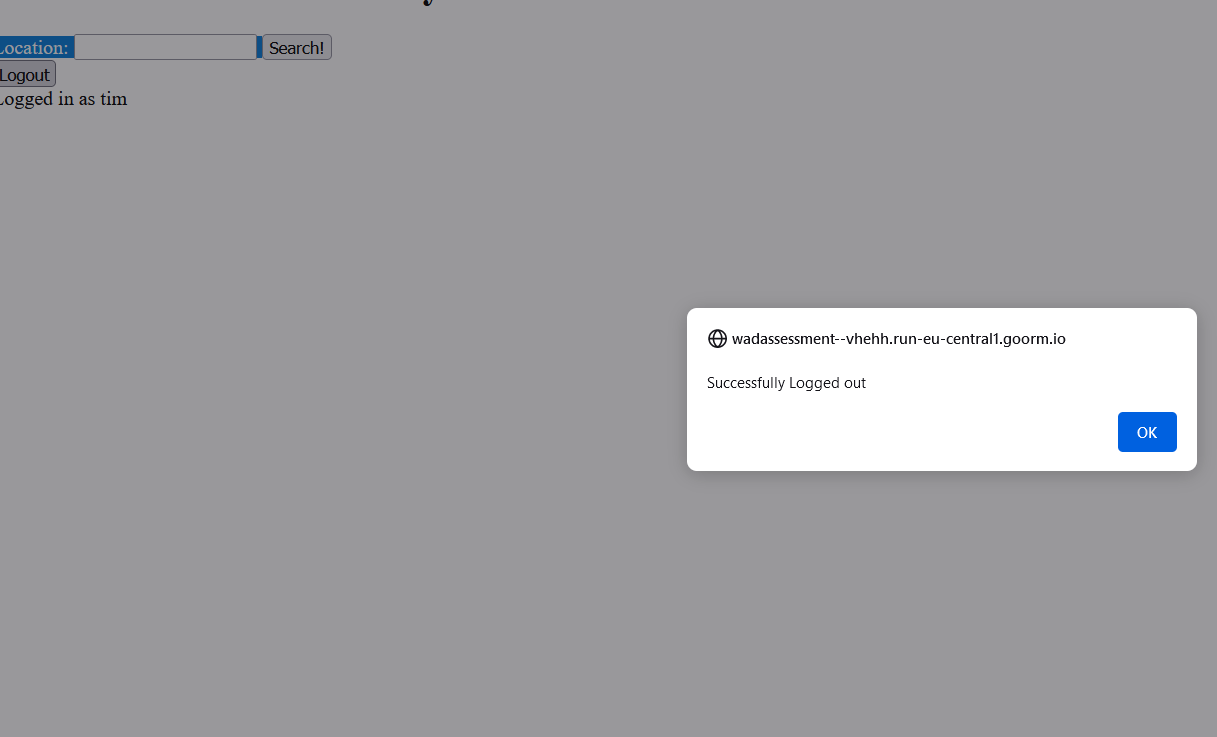
}

});

This part of the code runs without any problem.







Question 9: user must be logged-in to book accommodation.

For this purpose, is needed to add a middleware before book route. Then the middleware will be run before the book route. Usermiddleware.js is a file that keeps the middleware module. The usermiddleware function, checks req.session.username, if it is null then it sends an error message as a respond otherwise it calls the next().

*//user middleware in a seperate file into a funtion*

const dotenv = require('dotenv').config();

function usermiddleware(req, res, next) {

if (!req.session.username) {

*// process.env.APP\_USER does not exist (it's undefined)*

*// Return a 401 (Unauthorized) HTTP code, with a JSON error message*

res.status(401).json({ error: "You're not logged in. Go away!" });

} else {

next();

}

}

module.exports = usermiddleware;

Then in the main.js, before book route the middleware will be called for the book route.

const usermiddleware = require('./usermiddleware');

app.use('/acc\_bookings', usermiddleware);

The last step is changing the message in index.js to showing to the user.

btn.addEventListener('click', async (e) => {

const response = await fetch(`/acc\_bookings/book`, {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify(acc),

});

if (response.status == 200) {

var text2=`Accomodation with "ID:${accomodation.ID}" "Name:${accomodation.name}" Successfully Booked!`

alert(text2);

}else if (response.status==401) {

alert('You have not logged in!');

}else if (response.status==500) {

alert('server error');

}

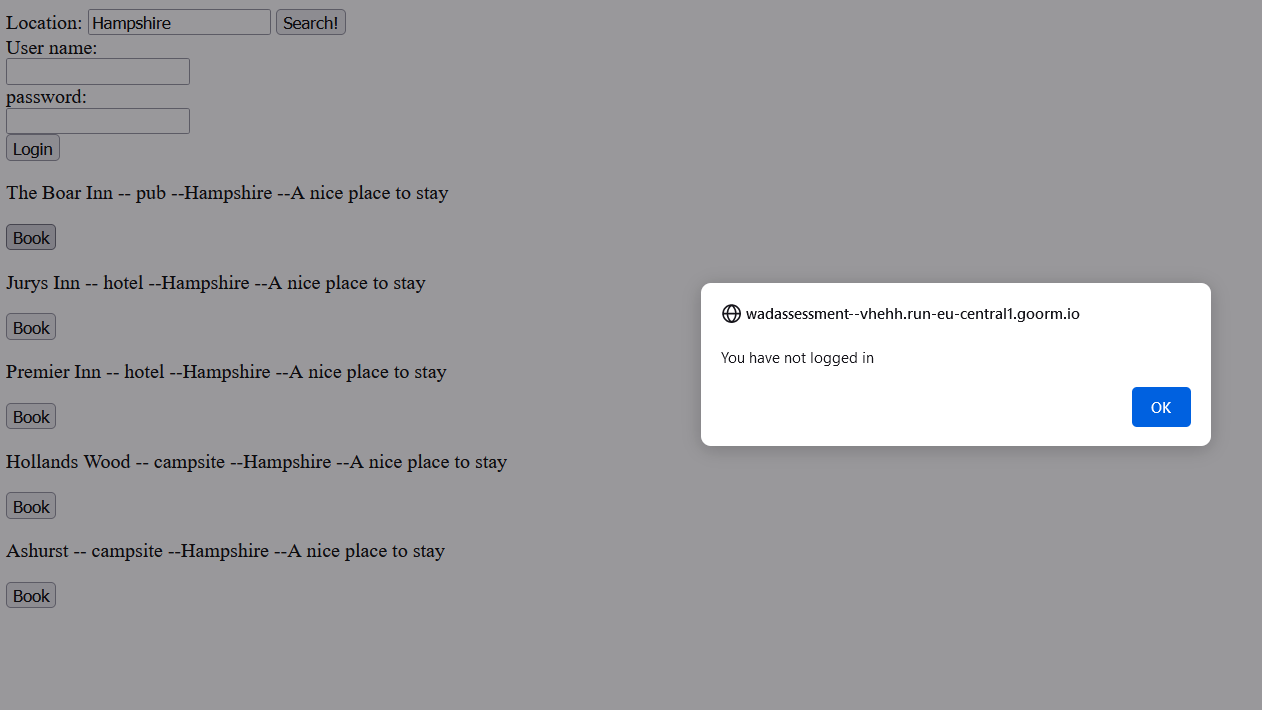
*// Parse the JSON.*

*//const products = await response.json();*

});

});

This part of the code runs without any problem.



Part F:

Question 10: Unfortunately, I couldn't finish Part F, it had a few bugs and I decided to not doing it.

Question 11: Add a booking button to your popup from Task 7 to allow the user to book that item of accommodation.

To doing this is needed to change index.js using DOM. first we create a div and text element. We add information of accomodation to the Text NODE. Then we append the Text as a child to the dive we already created.

const domDiv = document.createElement('div');

const Text = document.createTextNode(`Name:${accomodation.name}" "Description:${accomodation.description} `);

*//var text2=` "Name:${accomodation.name}" "Description:${accomodation.description}"`*

domDiv.appendChild(Text);

Then we add a label and a textbox for number of people, and we add them to domDiv and a break line at the end.

const label1=document.createTextNode("Please add number of people");

domDiv.appendChild(label1);

*//adding a textbox*

var np = document.createElement('input');

np.setAttribute('type', 'Text');

np.setAttribute('id', 'nptext');

domDiv.appendChild(np);

*//break line*

var BR = document.createElement('BR');

domDiv.appendChild(BR);

Then we create add a label and a date element to domDIv.

const label2=document.createTextNode("Please add the date you wish to book in format(yyyy/mm/dd) ");

domDiv.appendChild(label2);

*//adding date module*

var BR = document.createElement('BR');

domDiv.appendChild(BR);

*//adding a textbox var now = new Date();*

var now = new Date();

var date = document.createElement('input');

date.setAttribute('type', 'date');

date.setAttribute('id', 'datetext');

domDiv.appendChild(date);

*//break line*

var BR = document.createElement('BR');

domDiv.appendChild(BR);

At the end, is just needed to add a book button and add the whole domDIV to the popup.

*//adding booking button*

var btn = document.createElement('input');

btn.setAttribute('type', 'button');

btn.setAttribute('value', 'Book');

domDiv.appendChild(btn)

*//adding the whole domDIV to popup*

const marker = L.marker(pos).addTo(map);

marker.bindPopup(domDiv);

Then into the event listener is needed to read entered value by user and fill the jason with those values.

var npeople=document.getElementById('nptext').value;

var thedate=document.getElementById('datetext').value;

const date2 = new Date(thedate);

var thedate=`${date2.getFullYear()-2000}${date2.getMonth()+1}${date2.getDate()}`;

This part also runs without any problem.

Conclusion:

The assessment brief was clear and understandable. Everything has covered by Nick during the course. I am glad about the progress I had about client/server programming. The important fact for me was to learn all the concepts and I could get that with all helps from Nick. Because of lack of time, I couldn't do 100% the assessment but it was just because of time not because of any problem in understanding the concepts as Nick's methods in teaching is great. Thank you for everything.