**Details**

**ไฟล์ที่ต้องการ**

* dataAPI.js (to start the API node server)
* postGISConnection.js  (to connect to the centennial database)
* index.html (main app)
* app.js (to start the app node server)
* dataAPI.test.js (the final file that the code passes the test)
* geoJSON.js
* GeoJSON.test.js (the final file that the code passes the test)
* jsdoc.conf.json (to JSDoc all functions)
* locationServices.js
* simpleJest.test.js (the final file that the code passes the test)
* etc.

\*\*Title at the top of the website: **UCESSUV ucessuv**

<title><b>UCXXXXX<br>ucxxxxx</b></title>

1. **Location based services to be active on mobile devices, i.e. on smaller screens**

The index.html file provided earlier in the module has a sub menu called *Screen Size Check* which contains HTML menu option set up to show different screen sizes.

Create *locationServices.js (*reference this file from your *index.html* file)

"use strict"

/\*\*

\* locationTrackerId

\*

\* stores the ID of the location tracker so that it can be switched off if necessary

\* avoids having multipole location services running at the same time

\*

\*/

let locationTrackerId;

* Create a new menu group called *Location Tracking*that only appears when the screen size is medium or below.

<!-- menus that change with scren size -->

<div class="nav-item dropdown d-sm-down">

<a href="#" class="nav-link dropdown-toggle" data-bs-toggle="dropdown"><i class="fa fa-laptop me-2"></i>Screen Size Check</a>

<div class="dropdown-menu bg-transparent border-0">

<!-- only display when screen width is medium -->

<!-- default with block is that everything above is also visible so need to use lg-none to stop that -->

<a class="dropdown-item d-none d-md-block d-lg-none" href="#" onclick="alert('d-none d-md-block d-lg-none')"></i>Only medium</a>

<!-- only display when screen width is medium or below -->

<!-- when you use none, everything below is VISIBLE, everything above is not -->

<a class="dropdown-item d-lg-none" href="#" onclick="alert('d-lg-none')"></i>Medium or below</a>

<!-- only display when screen width is medium or above -->

<!-- d-xs-none does not exist, use d-none -->

<a class="dropdown-item d-sm-none d-none d-md-block" href="#" onclick="alert('d-sm-none d-none d-md-block')"></i>Medium or above</a>

<!-- only display when screen width is NOT medium -->

<!-- default when using none is that everything below is visible, everything above is not

so in this case you also need to say specifically that everything above is visible using d-lg-block -->

<a class="dropdown-item d-md-none d-lg-block" id="notmedium" href="#" onclick="alert('d-md-none d-lg-block')"></i>Not medium</a>

<!-- only display when screen width is large or above -->

<a class="dropdown-item d-sm-none d-md-none d-none d-lg-block" href="#" onclick="alert('dropdown-item d-sm-none d-md-none d-none d-lg-block')"></i>Large or above</a>

* work on the above example, add two sub menus *Swtich Location Tracking On* and *Switch Location Tracking Off*
* we'll use a call with a parameter to switchLocationTrackingOn - this is because we want to have different options in future for what happens when location tracking is enabled.   Your code should look like this:

<a class="dropdown-item d-lg-none" href="#" onclick="switchLocationServicesOn('showPosition')"></i>Switch Location Tracking On</a>

* This passes a string parameter which is the name of the function that we want to call when location tracking is activated.  By changing the parameter, we can change what happens.
* Add two functions with alert messages *swtichLocationServicesOn* and *switchLocationServicesOff*
* Add this code to the *swtichLocationServicesOn* function

/\*\*

\* @function swtichLocationServicesOn

\*

\*

\* @param locationFunction string - the function that is called when the locaiton tracking is swtiched on

\*

\* @description start location services - make sure that there is only one location service running at a time

\* <BR><br>having locationFunction as a parameter means that we can vary what happens with the tracking information

\*

\*

\*/

function switchLocationServicesOn(locationFunction) {

// we need to check if the location service is already in use, and if it is remove it before creating a new one

// so that we don’t have multiple tracking going on

if (navigator.geolocation) {

try {

(navigator.geolocation.clearWatch(locationTrackerId));

}

catch (e){

console.log(e);

}

// need to tell the tracker what we will do with the coordinates – showPosition

// also what we will do if there is an error – errorPosition

// also set some parameters – e.g how often to renew, what timeout to set

// timeout - how long the system will keep trying for a location before rasing an error

// maximumAge - how long to cache the position

const options = {

enableHighAccuracy: true ,

maximumAge: 5000,

timeout: 27000

};

// start the new location tracking service

// showPosition is the function that processes location information once it is obtained

// errorPosition is the function that is called if there is an error

// options are the settings above - height accuracy, how frequneelty position is measured

// we use window[locationFunction] here as we receive the function name as a STRING from the menu but need to CAST (convert) it to a function call (strings are just text)

locationTrackerId = navigator.geolocation.watchPosition(window[locationFunction],errorPosition, options);

}

else {

alert("Location tracking not supported on this device");

}

}

* Add the function that processes the location information
* "coordinates": [37.117020, -84.061539]

/\*\*

\* @function showPosition

\*

\* @description update the user's location on the web page

\*

\* @params position - the location information, derived from the watchPosition command

\*

\*/

function showPosition(position) {

console.log("you have moved");

document.getElementById('clickCoordinates').innerHTML = " Lat: " + position.coords.latitude + " Lng: " + position.coords.longitude +" Horizontal Accuracy: "+ position.coords.accuracy + " Altitude Accuracy: "+position.coords.altitudeAccuracy+" Heading: "+position.coords.heading+" Speed: "+position.coords.speed+" Altitude: "+position.coords.altitude;

}

* For error handling, we'll try an alert message.

/\*\*

\* function errorPosition

\*

\* if there is an error in the location tracking service, run this code

\*

\*/

function errorPosition(error){

console.log(error);

}

* Add the following code to the *switchLocationServicesOff* function.

// make sure to ues the locationTrackerId variable so that the tracker is the one that was created

// use a try/catch statement just in case locationTrackerId has not yet been used (i.e. it would be undefined and cause an error)

try {

navigator.geolocation.clearWatch(locationTrackerId);

}

catch (error){

// no need to do anything if there is an error

}

// clear the DIV that contains the tracking information

document.getElementById('clickCoordinates').innerHTML = "";

* Test on your mobile phone

A screenshot of a phone

Description automatically generated

* Test on Browser

A screenshot of a computer

Description automatically generated

* Create a new menu option under the *Location Tracking* menu called *Distance from Point*
* Add the following code to *locationServices.js*

/\*\*

\* @function calculateDistance

\*

\* @description given coordinates and a unit, calculate the distance between the points

\* code adapted from https://www.htmlgoodies.com/beyond/javascript/calculate-the-distance-between-two-points-in-your-web-apps.html [accessed 24 Dec 2023]

\*

\* @param {double} lat1 - the latitude of the first point

\* @param {double} lon1 - the longitude of the first point

\* @param {double} lat2 - the latitude of the second point

\* @param {double} lon2 - the longitude of the second point

\* @param {String} unit - the unit for disance - by deafult the distance is in miles, using K gives kilometers, and N gives nautical mines

\*

\* @returns {number} dist - the distance between the provided coordinates

\*/

function calculateDistance(lat1, lon1, lat2, lon2, unit) {

let radlat1 = Math.PI \* lat1/180

let radlat2 = Math.PI \* lat2/180

let radlon1 = Math.PI \* lon1/180

let radlon2 = Math.PI \* lon2/180

let theta = lon1-lon2

let radtheta = Math.PI \* theta/180

let subAngle = Math.sin(radlat1) \* Math.sin(radlat2) + Math.cos(radlat1) \* Math.cos(radlat2) \* Math.cos(radtheta);

subAngle = Math.acos(subAngle)

subAngle = subAngle \* 180/Math.PI // convert the degree value returned by acos back to degrees from radians

let dist = (subAngle/360) \* 2 \* Math.PI \* 3956; // ((subtended angle in degrees)/360) \* 2 \* pi \* radius where radius is 3956 miles

if (unit=="K") { dist = dist \* 1.609344 } // convert miles to km

if (unit=="N") { dist = dist \* 0.8684 } // convert miles to nautical miles

return dist

}

* Create *distanceFromPoint* function, add the following code. Note the quotation marks around the call to *switchLocationServicesOn* - this string is the name of the function that is called when location services are activated.

/\*\*

\* @function distanceFromPoint

\*

\* @description start the process to calculate the distance between the user's current location and a given fixed point

\*

\*

\*/

function distanceFromPoint() {

// switch location services on

// and then for the return function that is activated once location services are on

// call calcDistanceFromFixedPoint

switchLocationServicesOn("calcDistanceFromFixedPoint");

}

* Add the code that is called when the location service is activated.

/\*\*

\* @function calcDistanceFromFixedPoint

\*

\* @description complete the process to calculate the distance between the user's current location and a given fixed point

\* <br>and add a point to the map if it is below 500m

\*

\* @param position - the latitude and longitude data supplied by the navigator

\*

\*/

function calcDistanceFromFixedPoint(position){

// fixed point - the approximate location of UCL

let lat = 51.522449;

let lng = -0.132630;

let userLat = position.coords.latitude;

let userLng = position.coords.longitude;

// switch location tracking off as otherwise the function is called repeatedly

switchLocationServicesOff();

let distance = calculateDistance(lat, lng, userLat, userLng, "K");

// remember, distance units are in km

console.log(distance);

if (distance < 0.5) {

// create a marker

let testMarkerOrange = L.AwesomeMarkers.icon({

icon: 'play',

markerColor: 'orange'

});

// add the point to the map

let myLocation = L.marker([position.coords.latitude,position.coords.longitude], {icon:testMarkerOrange}).bindPopup(position.timestamp);

myLocation.addTo(mymap);

// zoom to the user's location

mymap.flyTo([position.coords.latitude, position.coords.longitude],12);

}

}

1. **Creating a Simple NodeJS Express Server**

* Create a branch called *express*
* Create a file called dataAPI.js, add the following code into the file.

"use strict";

/\*\* Express router providing user related routes

\* @module dataAPI

\* @requires express

\* @requires path

\*

\*/

/\*\*

\* express module

\* @description express is the server that forms part of the nodejs program

\* @const

\*/

const express = require('express');

/\*\*

\* path module

\* @description path is the module that logs the requests made to the router

\* @const

\*/

const path = require("path");

/\*\*

\* Express router to mount user related functions on.

\* @type {object}

\* @const

\* @namespace dataAPI

\*/

const dataAPI = express();

* Add some code to actually create the server.   Note that we are assigning the server a fixed PORT number - this is mapped back to the https://<<your server name>>/api link that is set up in Apache on the server.

// add an http server to serve files

let http = require('http');

let httpServer = http.createServer(dataAPI);

let port = 4480;

let server = httpServer.listen(port);

* Add some code to allow CORS - this is Cross Origin Resource Sharing.   What that means is that we can server data from https://<<your server name>>/api  and combine it with data or HTML or code from another website (in our case https://<<your server name>>/api

// adding functionality to allow cross-domain queries

dataAPI.use(function(req, res, next) {

res.setHeader("Access-Control-Allow-Origin", "\*");

res.setHeader("Access-Control-Allow-Headers", "X-Requested-With");

res.setHeader('Access-Control-Allow-Methods', 'GET,PUT,POST,DELETE');

next();

});

* Add the code to send a basic response when the user types in https://<<your server name>>/api

/\*\*

\* Route serving basic test message

\* @name /

\* @function

\* @memberof module:dataAPI

\* @inner

\*/

dataAPI.get('/',function (req,res) {

res.send("hello world from the Data API on port: "+port);

});

* Add some code to log all the requests to console.log so that you can track what is happening on the server.

// adding functionality to log the requests

dataAPI.use(function (req, res, next) {

let filename = path.basename(req.url);

let extension = path.extname(filename);

console.log("The file " + filename + " was requested.");

next();

});

* Upload the server and add it to GitHub and commit and push your changes.  The dataAPI.js file should go in the following directory - i.e. the root directory of your GitHub API code.

1. **Creating Automated Unit Tests Using JEST**

* Pull/merge all the previous API code: express --> main branch
* Create a branch called *jest*
* Clone the code to your server
* Add JEST as the test enviornment in package.json scripts, as follows *npm install jest --save*
* Create a new file called *simpleJest.test.js*, add the following text to the file - this is a simple, hard coded test that we can use to demonstrate that JEST is working as expected

"use strict";

test("string matches",() => {

// test for match the string - Success

let string1 = "First test for cege0043"

expect(string1).toMatch(/cege0043/);

// test for not match the string - this test will also

// pass as not.toMatch is correct in this case

expect(string1).not.toMatch(/abc/)

})

* Add another test to simpleJest.test.js as follows.

test("string does not match",() => {

// test for match the string - Success

let string2 = "Second test for cege0043"

expect(string2).toMatch(/second/);

})

* This test is a test that will fail.

A computer screen shot of a program

Description automatically generated

* Adapt the above testing code to fix the problem so that the test passes.
* JEST has the ability to start a web server, run some tests, and then stop the web server.  This is achieved via the *[supertest](https://www.npmjs.com/package/supertest)* package.
* On your server, install the following package: *npm install supertest –save*
* Create a new file called dataAPI.test.js
* Add the following code into the file - this references the required packages for the test

'use strict';

const supertest = require('supertest');

const request = supertest('http://localhost:4480');

* Now write the code to run the test - make sure the string matches exactly what you've put in the dataAPI.js file

test('returns a welcome message', async () => {

const response = await request.get('/');

expect(response.status).toBe(200);

expect(response.text).toEqual("hello world from the Data API on port: 4480");

});

* Run the test as follows

*./node\_modules/jest/bin/jest.js --verbose --detectOpenHandles ./js/dataAPI.test.js*

* Add a new test called *reverses the text* to the dataAPI.test.js file.  This test should:
  + Call an API end point /*reverseText?texttoreverse=xxxxx xxxxx*
  + Check that the response status of the return is 200
  + Check that the text has actually been reversed correctly
  + Where *xxxxx xxxxx* is any text that you want to use for testing.
* Save the text, upload your code to the server
* Run the test as above  - the test should fail
* Add the code into dataAPI.js.  Note that this code sample is deliberately incorrect so that you can use a test driven development approach to update the code until the above test passes.  To fix the bugs, you may need to think about:
  + how endpoints work
  + [how arrays work in JavaScript](https://www.w3schools.com/js/js_arrays.asp) (https://www.w3schools.com/js/js\_arrays.asp)
  + how we are sending status values to the browser with the result from any request to the API

/\*\*

\* Route serving basic test message

\* @name /reversetext

\* @function

\* @memberof module:dataAPI

\* @inner

\* @param {string} - the string to be reversed

\*/

dataAPI.get('/reversetext',function (req,res) {

//note, this is the buggy version for the students to fix

let originalText = req.query.texttoreverse;

console.log(originalText);

// convert it to an array - we can use the built in Array.from for this

let textArray = Array.from(originalText);

console.log(textArray.length);

let reversedText = "";

// loop over the array backwards

// here i > 0 is the condition that, while true, will keep the loop

// going - as soon as i > 0 returns false, the loop stops

for (let i = textArray.length; i > 0; i--){

console.log(i);

reversedText = reversedText + textArray[i];

}

// finally send the result back to the browser

res.send(reversedText);

});

* Run the test again. The code will fail. Fix any bugs, re-running the test each time until the code passes the test.
* Extend the *reverses the text*  test to include the following situations:
  + The text contains spaces in the middle
  + The text contains a space at the beginning
  + The text contains a space at the end
  + The text contains a % sign
  + The text contains a $ sign
  + The text contains an &
  + The text contains a \*
  + You may find [URL Encoding](https://www.w3schools.com/tags/ref_urlencode.asp) relevant for these tests
* Now that the tests work locally, you can use GitHub Actions to test automatically. We already have a simple testing file call js.yml, in the .github/workflows directory.  You first need to edit this file so that the API is started before the tests are run.
* Find the js.yml file, which is in the following directory (note that there is a . before gitub)

/home/<<your cs username >>/code/<<your github api repository name>>/.github/workflows

* In a text editor, edit the file so that the final text looks as follows:

name: Javascript Unit Tests

on:

push:

branches:

pull\_request:

branch: [main]

workflow\_dispatch:

jobs:

Jest:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v1

- name: Run Tests

run: |

npm install

node dataAPI &

npm run test

#

* Save the file on your server, and commit
* Push the code to GitHub.
* Go to the GitHub website and check that the action has run without errors.  If there are any errors drill down into the action to see the cause.

1. **PostGIS Connectivity (1)**

* Keep working in the branch from the previous week
* At command line, go to the root directory of your API code and type the following to install the PostgreSQL / PostGIS package that contains all the code we need to connect to the database: *npm install pg --save*
* Create a file called *geoJSON.js*
* Add the following code at the top of the file - this code calls all the external Node modules we will need for our route

"use strict";

let express = require('express');

let geoJSON = require('express').Router(); // create a new route

let pg = require('pg'); // the code to connect to PostgreSQL

let fs = require('fs'); // code to read the database connection details file

* Add a test end point

geoJSON.route('/testGeoJSON').get(function (req,res) {

res.json({message:req.originalUrl});

});

* At the very bottom of the file, add the following code

module.exports = geoJSON;

* Upload the geoJSON.js file to a sub-directory called *routes* - i.e. to the following structure

/home/<<your CS username>>/code/<<your GitHub API repository name >>/routes

* We now need to link from the main express server to this one, so that if this new route is passed to the web server as a request it will be directed correctly.
* At the bottom of *dataAPI.js*, add the following

const geoJSON = require('./routes/geoJSON');

dataAPI.use('/geoJSON', geoJSON);

* Test your code - you should use the following URL, adapted to your new route name https://<< your CS computer name >>/api/<<new route name>>/testGeoJSON?name=claire&surname=ellul

1. **PostGIS Connectivity (2)**

* Create a file called *postGISConnection.js*
* Add the following content to the file, in exactly the order shown - make sure to use your own username and password, and make sure you include the commas as shown
  + host: cege0052.cege.ucl.ac.uk,
  + user: user798,
  + database: user798db,
  + password: cege0052llh798,
  + port: 5432
* Upload it to your server and add it to the following directory

/home/<< your CS username>>/certs

* Add the following code to the geoJSON.js file, just above the *testGeoJSON* end point. This code will retrieve your username from the Linux operating system, so that the code can work out where the connection details file is stored

// we need to get the user's name from the login used

// to connect to the server

// that way we can work out the correct path for the connection file

// /code/<<username>>/certs/

const os = require('os');

const username = os.userInfo().username; // locate the database login details

console.log(username);

* Read the file using the fs (file system) package  - the path to the file includes the user's login name from the previous step

// oad the database login details file

let configtext = ""+fs.readFileSync("/home/"+username+"/certs/postGISConnection.js");// locate the database login details

* Loop through the file line by line, extract the login information, create a configuration and start the database pool

// now convert the configruation file into the correct format -i.e. a name/value pair array

// this means looping through the file looking for commas

// each comma indicates a new line, a new piece of information

// we then take the information and convert it into a configuration

// for the PostgreSQL connection

let configarray = configtext.split(",");

let config = {};

for (let i = 0; i < configarray.length; i++) {

let split = configarray[i].split(':'); //split = split one text string into two

config[split[0].trim()] = split[1].trim(); //trim = remove any spaces before or after the text

}

let pool = new pg.Pool(config);

console.log(config);

* Add the following code underneath the previous testGeoJSON  end point.  This will connect to the database and - if the connection works - run the SQL and return the result  to the browser.

geoJSON.get('/postgistest', function (req,res) {

// create a new connection in the pool

// the connection will return err if it failes

// if it works, the connection will return a client called client - which can be

// used to run some SQL

// done is the name of the function to be called once the SQL has

// returned a value - this closes the connection so that it can be

// reused

pool.connect(function(err,client,done) {

if(err){

console.log("not able to get connection "+ err);

res.status(400).send(err);

}

// the SQL that we want to run

let query = "select \* from information\_schema.columns";

// pass the SQL to the client from the pool

// will return err if it fails

// result will hold any values returned by the SQL

client.query(query ,function(err,result) {

done();

if(err){

console.log(err);

res.status(400).send(err);

}

// send to send the result back to the browser

// result.rows will give an array of all the rows

// in the result

res.status(200).send(result.rows);

});

});

});

* You can test your code using the following URL

https://<<your CS machine name>>/api/<<route name>>/postgistest

You will see a long list of the tables and schemas in the database.

**\*Testing Your PostgreSQL/PostGIS Connection - Checklist**

Here's a check list for if your database connection doesn't work ..

• Is the connection data correct in your postGISConnection.js file? E.g. is the IP address of the hostname correct, are you connecting to the ucfscde database?

• Is that file located in the correct location

• Is the file parsing correctly when you start the dataAPI.js server?

• Can you create a connection pool or do you get an error message at the console?

• Can you run the SQL you generate? Test this by using console.log to echo out the SQL and copy/paste into PGAdmin

1. **Create and End Point for Asset Information Retrieval**

* **Step 1\_1 - Create the JEST Test**

The functional requirement for our first end point is :

*Retrieve all the data from the cege0043.asset\_information*table as a GeoJSON feature collection.

1.  Create a new file called GeoJSON.test.js

2.  Add the following code at the top of the file, to reference the packages we need to run the tests

'use strict';

const supertest = require('supertest');

const request = supertest('http://localhost:4480');

const fs = require('fs');

3. Now start creating the first test, with the following details:

* test name: 'retrieve asset information'
* end point:  /<<your geoJSON route - see moodle >>/asset\_information
* status: 200

4.  The challenges we have are:

* is identifying what the response.text value should be - in theory it should be the GeoJSON data for the entire dataset, as that is what the functional requirement states.
* we also need to make sure that the dataset is returned in a specific order every time, as otherwise it won't match  - a database can return rows in random order - we can use an order by query for this when we implement the code

We could put the resuting data in as a long text string in the test file, but it is better to reference an external file that just contains the GeoJSON data.   You can find the file on Moodle, called *asset\_information.geojson*

5. Upload the file to the same directory as you have the tests in - i.e. the /js directory

6. Add the following code to the test to read the file and compare with the result of the API query

// we use readFileSync here so that the system waits

// until the full file is loaded

// an asychronous approach would mean that the code would try and run

// the next line of text before the file is loaded

// the ./ means look in the same directory as this file

// we use JSON.parse to convert the data into JSON for comparison

let asset\_information = JSON.parse(fs.readFileSync("./js/asset\_information.geojson"));

expect(JSON.parse(response.text)).toEqual(asset\_information);

* **Step 1\_2 - Run the Test**

1. Use the following command to run the JEST test you have just created

./node\_modules/jest/bin/jest.js --verbose --detectOpenHandles ./js/geoJSON.test.js

* **Step 1\_3 - Create the End Point**

1.  Open the geoJSON.js file and add the following code to retrieve the required GeoJSON data.

geoJSON.get('/asset\_information', function (req,res) {

pool.connect(function(err,client,done) {

if(err){

console.log("not able to get connection "+ err);

res.status(400).send(err);

}

let querystring = " SELECT 'FeatureCollection' As type, array\_to\_json(array\_agg(f)) As features FROM ";

querystring = querystring + "(SELECT 'Feature' As type , ST\_AsGeoJSON(st\_transform(lg.location,4326))::json As geometry, ";

querystring = querystring + "row\_to\_json((SELECT l FROM (SELECT id, asset\_name, installation\_date, user\_id, timestamp) As l )) As properties";

querystring = querystring + " FROM cege0043.asset\_information As lg order by id limit 100 ) As f";

console.log(querystring);

client.query(querystring,function(err,result) {

done();

if(err){

console.log(err);

res.status(400).send(err);

}

else {

res.status(200).send(result.rows);

}

});

});

});

* **Step 1\_4 - ReRun the Test**

1. Run the test again - does it pass?

NB: This test assumes that the data is STATIC - if someone makes a micro change to a tiny element of the data in the database - specifically to the first 100 records as we have a *limit 100* statement -  the test will automatically fail.

* **Step 2 - Create a Generic GeoJSON Retrieval API End Point**

The functional requirement for this end point (which will be called /*getGeoJSON*) is

Given a schema, table name, the name of the ID column and the name of the column that contains the geometry, return the GeoJSON for the data in the table

* **Step 2\_1 Write the JEST Test**

It is much more tricky to write a JEST test when we don't actually know the data that will be retrieved.  However, there are some properties of [GeoJSON](https://datatracker.ietf.org/doc/html/rfc7946) that we could use to indicate that the response is correct.

* Is the response actually valid JSON
* We can also test if the response valid GeoJSON
* Is the first level of the response a feature collection
* Is there one or more features in the feature collection
* Does each feature have geometry

Finally, we can test if the response does in fact correspond to data from the table we are querying

* Do the properties of the feature correspond to the column names of the table (we can use the information\_schema to get these)

We will create all these tests separately to help ensure that the code is readable.  All tests should be inserted into the geoJSON.test.js file

1. Create a test to confirm that the result is valid JSON

test('is valid JSON', async () => {

const response = await request.get('/geoJSON/getGeoJSON/cege0043/asset\_information/id/location');

expect(response.status).toBe(200);

let isJSON = false;

try {

JSON.parse(response.text);

isJSON = true;

} catch (e) {

isJSON = false;

}

expect(isJSON).toEqual(true);

});

2. Create a test to confirm whether the response is GeoJSON - first we need to define a function that will take JSON data, and parse it (loop through it) to look for properties, type is feature and geometry

/\*\*

\* @function

\* @description test to see whether the returned JSON data is

\* <br> valid geoJSON - by looking for three key names

\* <br> geometry, type, properties

\* <br> note that we have to dig down into the JSON array to get these

\* <br> values

\* @param {JSON} geoJSON - the geoJSON data to be tested

\* @returns {boolean} - true if the test passed, false if it failed

\*/

function testForValidGeoJSON(geoJSON){

let isGeoJSON;

try {

Object.keys(geoJSON).forEach(function(key) {

//console.log('Key : ' + key + ', Value : ' + geoJSON[key])

let subGeoJSON = geoJSON[key];

Object.keys(subGeoJSON).forEach(function(key1){

//console.log('Key1 : ' + key + ', Value1 : ' + subGeoJSON[key1])

let subsubGeoJSON = subGeoJSON[key1];

Object.keys(subsubGeoJSON).forEach(function(key2){

//console.log('Key2 : ' + key2 + ', Value2 : ' + subsubGeoJSON[key2])

// at this level, the key should be one of

// geometry

// properties

// type - which should be feature

// if not this is not valid GeoJSON

if (key2 == 'properties' || key2=='geometry'|| (key2=='type'&& subsubGeoJSON[key2]=='Feature')){

isGeoJSON = true;

}

else {

// no need to continue

return false;

}

})

})

})

} //end of the try

catch (e){

console.log(e.message);

// an error has occured so return false

return false;

}

// if we got to here, then the test has passed

return isGeoJSON;

}

3. Now call this function from a test

test('is geoJSON', async () => {

const response = await request.get('/geoJSON/getGeoJSON/cege0043/asset\_information/id/location');

expect(response.status).toBe(200);

let isGeoJSON = false;

let geoJSON = JSON.parse(response.text);

isGeoJSON = testForValidGeoJSON(geoJSON);

expect(isGeoJSON).toEqual(true);

});

4.  Create a test to look specifically at the attribute names given in the properties - do these correspond to the correct column names for the table?  We can make use of the previous code we wrote - the /postgistest end point which returns all these details from the information schema.  Add the following function first, to get the list of property names from the GeoJSON.

/\*\*

\* @function extractColNames

\* @description parse geoJSON until the first instance of properties is found

\* <br> and use the keys in the properties data to get an array of

\* <br> column names in that geoJSON data

\* @param {JSON} geoJSON - the geoJSON data

\* @returns {Array} colNames - the array of column names

\*/

function extractColNames(geoJSON){

let colNames = []; // emptry array to store the column names

try {

let properties = geoJSON.features[0].properties;

// now loop around this and get the key values

Object.keys(properties).forEach(function(key) {

colNames.push(key);

});

} //end of the try

catch (e){

console.log(e.message);

// an error has occured so return false

return false;

}

// if we got to here, then the test has passed

return colNames;

}

Note: in this function we could have looped around the GeoJSON as we did previously.  However, we only need to find one set of properties here, whereas the previous function is testing ALL the GeoJSON file.

5. Now add a function to parse the information schema data

/\*\*

\* @function

\* @description take a list of all the tables and column names

\* <br> in our database, and loop through and find the column names that

\* <br> correspond to the given table and schema

\* <br> NB: we use schema as well as a table could appear with the same name

\* <br> in two different schemas

\* @param {String} tableName - the name of the table to check

\* @param {String} schemaName - the name of the schema to check

\* @returns {Array} informationSchema - the list of columns

\*/

function extractColNamesInformationSchema(infoCols,schemaName,tableName){

let infoColumns = []; //an empty array to store the results

// now loop through to find entries where

// table\_schema = the schema name and

// table\_name = the table name

// then get the column\_name

let infoColsJSON = JSON.parse(infoCols);

Object.keys(infoColsJSON).forEach(function(key) {

let infoSchema = infoColsJSON[key].table\_schema;

let infoTableName = infoColsJSON[key].table\_name;

if (infoTableName ==tableName && infoSchema ==schemaName) {

// get hold of the column name

infoColumns.push(infoColsJSON[key].column\_name);

}

});

return infoColumns;

}

6. Now add a function to compare the two lists

/\*\*

\* @function

\* @description - take an array of the matchingcolumn values from

\* <br> the information schema query and compare them to the

\* <br> property names in the GeoJSON file

\* @param {Array} colNames - the column names from the GeoJSON file

\* @param {Array} colNamesInformationSchema - the column names from the informatio

\* <br> schema query

\* @returns {Boolean} - true if all column names are found, false if not

\*/

function compareColNames(colNames,colNamesInformationSchema){

for (let i = 0;i<colNames.length;i++ ){

let foundAMatch = false;

for (let j= 0;j<colNamesInformationSchema.length;j++){

if (colNames[i] == colNamesInformationSchema[j]){

foundAMatch = true;

// break out of the loop

j = colNamesInformationSchema.length;

}

}

// if we get to here and foundAMatch is false

// then we didn't find a match for that column

if (foundAMatch == false){

return false;

}

}

// if we get to here then match was found for all columns

return true;

}

  7. Finally, create the test itself - note that we make both API calls here as they need to be in an asynchronous function so that the code waits for the responses

test('colNames correct', async () => {

const response = await request.get('/geoJSON/getGeoJSON/cege0043/asset\_information/id/location');

const infoCols = await request.get('/geoJSON/postgistest');

let colNamesCorrect = false;

let geoJSON = JSON.parse(response.text);

let colNames = extractColNames(geoJSON);

let colNamesInformationSchema = extractColNamesInformationSchema(infoCols.text,'cege0043','asset\_information');

colNamesCorrect = compareColNames(colNames,colNamesInformationSchema);

expect(colNamesCorrect).toEqual(true);

});

* **Step 2\_2 Test the JEST Tests**

1. Use the command line text above to test the tests - they should fail

* **Step 2\_3 Write the End Point**

For this end point we will pass the required parameters - schema, tablename, ID column name, geometry column name - as [template parameters](https://rapidapi.com/blog/api-glossary/parameters/) - i.e. they form part of the URL rather than being sent as name/value pairs after a ? symbol.   This is because we always need the four parameters for the query to work - and they are not optional.

1. Create a new end point in geoJSON.js, called *getGeoJSON*

*2. A*dd the following code to create the end point.   As a reminder, this code works in two stages

* We first use the information\_schema to find out what columns exist in the table.  These become the properties of the GeoJSON dataset
* We then combine this information with SQL to create a JSON Array - i.e. combine all the different rows in the table into one long GeoJSON feature collection

geoJSON.get('/getGeoJSON/:schemaname/:tablename/:idcolumn/:geomcolumn', function (req,res) {

pool.connect(function(err,client,done) {

if(err){

console.log("not able to get connection "+ err);

res.status(400).send(err);

}

let colnames = "";

// first get a list of the columns that are in the table

// use string\_agg to generate a comma separated list that can then be pasted into the next query

let tablename = req.params.tablename;

let schema = req.params.schemaname;

let idcolumn = req.params.idcolumn;

let geomcolumn = req.params.geomcolumn;

let querystring = "select string\_agg(colname,',') from ( select column\_name as colname ";

querystring = querystring + " FROM information\_schema.columns as colname ";

querystring = querystring + " where table\_name =$1";

querystring = querystring + " and column\_name <> $2 and table\_schema = $3 and data\_type <> 'USER-DEFINED') as cols ";

console.log(querystring);

// now run the query

client.query(querystring,[tablename,geomcolumn,schema], function(err,result){

if(err){

console.log(err);

res.status(400).send(err);

}

else {

let thecolnames = result.rows[0].string\_agg;

colnames = thecolnames;

console.log("the colnames "+thecolnames);

let cols = colnames.split(",");

let colString="";

for (let i =0; i< cols.length;i++){

console.log(cols[i]);

colString = colString + JSON.stringify(cols[i]) + ",";

}

console.log(colString);

//remove the extra comma

colString = colString.substring(0,colString.length -1);

// now use the inbuilt geoJSON functionality

// and create the required geoJSON format using a query adapted from here:

// http://www.postgresonline.com/journal/archives/267-Creating-GeoJSON-Feature-Collections-with-JSON-and-PostGIS-functions.html, accessed 4th January 2018

// note that query needs to be a single string with no line breaks so built it up bit by bit

// to overcome the polyhedral surface issue, convert them to simple geometries

// assume that all tables have an id field for now - to do add the name of the id field as a parameter

querystring = "SELECT 'FeatureCollection' As type, array\_to\_json(array\_agg(f)) As features FROM ";

querystring += "(select 'Feature' as type, x.properties,st\_asgeojson(y.geometry)::json as geometry from ";

querystring +=" (select "+idcolumn+", row\_to\_json((SELECT l FROM (SELECT "+colString + ") As l )) as properties FROM "+schema+"."+JSON.stringify(tablename) + " ";

querystring += " ) x";

querystring +=" inner join (SELECT "+idcolumn+", c.geom as geometry";

querystring +=" FROM ( SELECT "+idcolumn+", (ST\_Dump(st\_transform("+JSON.stringify(geomcolumn)+",4326))).geom AS geom ";

querystring +=" FROM "+schema+"."+JSON.stringify(tablename)+") c) y on y."+idcolumn+" = x."+idcolumn+") f";

console.log(querystring);

// run the second query

client.query(querystring,function(err,result){

//call `done()` to release the client back to the pool

done();

if(err){

console.log(err);

res.status(400).send(err);

}

else {

console.log(result.rows);

// the data from PostGIS is surrounded by [ ] which doesn't work in QGIS, so remove

// so we need to convert the JSON into a string temporarily

// remove the brackets and then convert it back when we send

// the result to the browser

let geoJSONData = JSON.stringify(result.rows);

geoJSONData = geoJSONData.substring(1);

geoJSONData = geoJSONData.substring(0, geoJSONData.length - 1);

console.log(geoJSONData);

res.status(200).send(JSON.parse(geoJSONData));

}

});

} // end error check from client

});

});

});

* **Step 2\_4 Test Again**

1. Rerun the tests, and debug your code until all the tests pass.

* **Step 2\_5  Optional: Add Additional Tests**

So far we've tested with one spatial table from our database.    Change the database in your database connection file to your own user database, and add some tests to test your own data from previous assignments.

Don't forget to change the database name back to ucfscde once you've run your own tests!

* **Step 2\_6  Optional: Test from the Client/Browser**

You can create new menu items to call the API and display the resulting GeoJSON data on the Leaflet map. When you are calling the API end point from the client make sure that you don't hard code the location of the server! Add the specific route, end point, schema, tablename, ID column and location column to the following string:

let URL = document.location.origin+"/api/"

* **Step 2\_7  Viewing API Documentation**

You can follow the same procedure to create API documentation as we did for the APP repository. However, we don't have a generic express router in the API - the API isn't designed to serve files, it is designed to serve JSON/GeoJSON. We need to add a specific end point to dataAPI.js to see the documentation, one that is restricted to files in the documentation directory.

dataAPI.use('/documentation', express.static(path.join(\_\_dirname, 'documentation')));

1. **Using JSDoc to Document Code**

* **Step 1 - Running the JSDoc Command**

Some of the sample code in the repository already has JSDoc-formatted documentation comments.   You need to compile (create) this documentation by installing and using the JSDoc code.

1. In GitHub, create a branch in the APP repository called *documentation*

2. Log into your server, and clone your code

3. At the command line and use the cd command to change directory to your repository.  Depending on where you are storing your code, the command will be something like

cd /home/<<your CS username>>/code/cege0043-apps-23-24-<<your github user name goes here>>

4. Check to see if the *documentation* directory already exists in your repository, by using the *ls* command to show the files in the repository directory.

5. If the documentation directory does not exist, then create it as follows

mkdir documentation

6. Use the following command to install JSDoc

npm install jsdoc --save

7. Make sure that the jsdoc.conf.json file exists by typing ls to see the content of the directory

8. Run the following command to create the documentation

./node\_modules/jsdoc/jsdoc.js -c ./jsdoc.conf.json --verbose

This command uses the instructions we have placed in the jsdoc.conf.json file to compile the documentation.  Those instructions specifically tell JSDoc to only create documentation for files with a .js extension - i.e. only document the JavaScript files, and also to exclude any files in *node\_modules* (as otherwise we'd be documenting all the NodeJS libraries we are using)

9. If the run is succesful, you will see messages as shown below

A screenshot of a computer screen

Description automatically generated

10.  The documentation can be accessed via a browser, by typing in the following URL

http://<<your CS computer name>>/app/documentation/index.html

11.  Documentation will appear as follows.  You can click on a function name on the list to see the detailed documentation for that function

A screenshot of a computer

Description automatically generated

12.  Every time you create documentation, new files will be added to the codebase.  Make sure you use git add -A ,  git commit -am "xxxx" and git push to save the files to GitHub.

* **Step 2 - Creating Documentation for a Simple Function**

Use the @function and @description tags to create the required documentation.  This example also shows how you can use HTML tags (in this case <br> for a line break) to format the documentation

/\*\*

\* @function showLeaflet

\*

\* @description show the div with the cesium map loaded

\* <br>no close button at the top as the user can swap back to 2D from the menu

\*

\*/

function showLeaflet() {

showDiv('mapWrapper');

}

* **Step 3 - Creating Documentation for a Function with Parameters**

The @param tag allows you to create specific references to the parameters that are required when calling a function).    I

It is important to specify the data type of the parameter as part of this doocumentation, using { } brackets.

There are a number of [standard parameter types](https://jsdoc.app/tags-type) in JSDoc that you can use - but the system also allows you to specifcy your own types as in this example.

/\*\*

\* @function

\* @param {GeoJSON} datasource - the geojson layer

\* @param {JSON} feature -the data with the style information. Includes the layer\_colour, the layer\_transparency

\* @description style the points added via geoJSON. Default outline colour of black is used.

\*/

function stylePoints(datasource,feature){

var entities = datasource.entities.values;

let fillColor = Cesium.Color.fromCssColorString(feature.properties.layer\_colour).withAlpha(feature.properties.layer\_transparency);

let outlineColor = Cesium.Color.fromCssColorString('black');

for (var i = 0; i < entities.length; i++) {

var entity = entities[i];

entity.billboard = undefined;

entity.point = new Cesium.PointGraphics({

color: fillColor,

outlineColor: outlineColor,

outlineWidth: 5,

pixelSize: 20

});

}

}

The above code also illustrates the flexibility of JSDoc - you don't need to specify the function name - the @function tag will tell JSDoc to get the function name from the next function declaration in the text.

* **Step 4 - Creating Documentation for a Function that Returns a Value**

The @return parameter allows you to specify the data type and description of values returned by a function.

/\*\*

\* @function calculateDistance

\*

\* @description given coordinates and a unit, calculate the distance between the points

\* code adapted from https://www.htmlgoodies.com/beyond/javascript/calculate-the-distance-between-two-points-in-your-web-apps.html [accessed 24 Dec 2023]

\*

\* @param {double} lat1 - the latitude of the first point

\* @param {double} lon1 - the longitude of the first point

\* @param {double} lat2 - the latitude of the second point

\* @param {double} lon2 - the longitude of the second point

\* @param {String} unit - the unit for disance - by deafult the distance is in miles, using K gives kilometers, and N gives nautical mines

\*

\* @returns {number} dist - the distance between the provided coordinates

\*/

function calculateDistance(lat1, lon1, lat2, lon2, unit) {

let radlat1 = Math.PI \* lat1/180

let radlat2 = Math.PI \* lat2/180

let radlon1 = Math.PI \* lon1/180

let radlon2 = Math.PI \* lon2/180

let theta = lon1-lon2

let radtheta = Math.PI \* theta/180

let subAngle = Math.sin(radlat1) \* Math.sin(radlat2) + Math.cos(radlat1) \* Math.cos(radlat2) \* Math.cos(radtheta);

subAngle = Math.acos(subAngle)

subAngle = subAngle \* 180/Math.PI // convert the degree value returned by acos back to degrees from radians

let dist = (subAngle/360) \* 2 \* Math.PI \* 3956; // ((subtended angle in degrees)/360) \* 2 \* pi \* radius where radius is 3956 miles

if (unit=="K") { dist = dist \* 1.609344 } // convert miles to km

if (unit=="N") { dist = dist \* 0.8684 } // convert miles to nautical miles

return dist

}