**Table of Contents**

[Table of Contents](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week7/practical7-step3.html#toc0)

[Creating a Dynamic GeoJSON End Point](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week7/practical7-step3.html#toc1)

[Step 1 - Create and End Point for Asset Information Retrieval](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week7/practical7-step3.html#toc2)

[Step 1\_1 - Create the JEST Test](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week7/practical7-step3.html#toc3)

[Step 1\_2 - Run the Test](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week7/practical7-step3.html#toc4)

[Step 1\_3 - Create the End Point](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week7/practical7-step3.html#toc5)

[Step 1\_4 - ReRun the Test](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week7/practical7-step3.html#toc6)

[Step 2 - Create a Generic GeoJSON Retrieval API End Point](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week7/practical7-step3.html#toc7)

[Step 2\_1 Write the JEST Test](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week7/practical7-step3.html#toc8)

[Step 2\_2 Test the JEST Tests](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week7/practical7-step3.html#toc9)

[Step 2\_3 Write the End Point](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week7/practical7-step3.html#toc10)

[Step 2\_4 Test Again](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week7/practical7-step3.html#toc11)

[Step 2\_5  Optional: Add Additional Tests](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week7/practical7-step3.html#toc12)

**Creating a Dynamic GeoJSON End Point**

In a previous practical, we saw how to create an end point that serves a list of the tables and schemas in the database.   However, as geospatial experts for the data to be useful we need it to be mappable. One way to acheive this is to create an API end point that serves GeoJSON data.

In this practical, we will first create an end point that serves GeoJSON from a single table.  We will then see how to create a dynamic end point that will create GeoJSON from any location-enabled table (i.e. table with a geometry column) in the database.

Before you being this practical, please make sure you have understood the step-by-step process that will be used to create GeoJSON Feature Collections from the database - see lecture slides

Remeber to change the routes to the routes in the sample code below to the routes that are listed on Moodle, otherwise your code won't work

Reminders:

* Make sure all the GitHub actions are switched off on your GitHub site (see previous practical for why)
* Make sure to commit and push your code frequently
* Make sure to include documentation for every end point you create - these are not provided for you, it is up to you to generate them
* Compile the JSDOC frequently
* The general procedure for unit testing using JEST is first write the test, then write the code, then test and fix bugs if necessary

**Step 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

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);

}

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

Remember to COMMIT frequently!

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;

}

6. 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);

}

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);

}

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));

});

});

});

});

**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!

Reminder: We can't run JEST tests on GitHub using Actions, as GitHub does not have access to our database or the database connection details in the *certs* directory.  So for this course, make sure your code passes JEST tests by running the tests yourself.