**Creating an EC2 PSQL Database**

**Security and Privacy**

First of all we need to create a security group. This will ensure that we can allow connections to and from our PSQL database. To do this go to security groups; VPC feature.

A blue rectangular object with a black border

Description automatically generated

Click on ‘create security group’.

A screenshot of a computer

Description automatically generated

**A screenshot of a computer

Description automatically generated**We now have a new security group created with a unique name. The unique name is managed by AWS. We now need to edit the rulesets.

**A screenshot of a computer

Description automatically generated**

Here we have created 3 separate rules. Note that we are accepting PostgreSQL connections from ALL ip addresses. In our ‘All traffic’ rule we have selected a local ip address of a machine that we want to use – this is more secure.

For the outbound rules we can leave the following configuration.

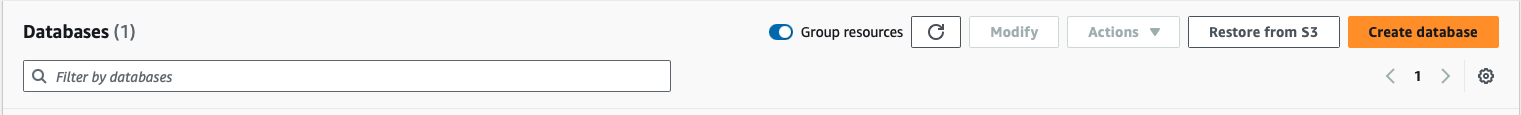
**A screenshot of a computer

Description automatically generated**

Now that we have created a security group, we can attach it to our PSQL instance.

**Creating the PSQL instance**

Go to Amazon RDS and select ‘Create database’.

****

Select standard create:

**A screenshot of a web page

Description automatically generated**

A screenshot of a computer

Description automatically generatedThen select the correct engine type. In this case we are using PSQL so use PostgreSQL:

The latest version is fine:

A screenshot of a computer

Description automatically generated

Select Free tier:

A screenshot of a computer

Description automatically generated

Create authentication settings:

A screenshot of a computer

Description automatically generated

Use db.t3.micro as the instance configuration:

A screenshot of a computer

Description automatically generated

Select default storage settings and ensure that storage autoscaling is set to off.

A screenshot of a computer

Description automatically generated

For connectivity settings ensure that **public access is set to yes**.

A screenshot of a computer

Description automatically generated

Make sure to attach the security group we created earlier.

A screenshot of a computer

Description automatically generated

Ensure that password authentication is on:

A screenshot of a computer

Description automatically generated

For additional configuration turn off encryption and ensure that there is an intial database name!

A screenshot of a computer

Description automatically generated

Finally click create database. It will take some time to setup, but eventually you will have a database setup.

A screenshot of a computer

Description automatically generated

In order to connect to the database you will need to make a note of some of the information.

A screenshot of a computer

Description automatically generated

**Connecting to the PSQL Database**

Now we can provide the information to make a connection to our PSQL server:

*const* fs = require('fs');

*const* format = require('pg-format');

*const* { Client } = require('pg');

// AWS Client Information:

*const* client = new Client ({

user: 'postgres',

host: 'database-5761i.cfk2gikqsjhw.eu-west-2.rds.amazonaws.com',

database: 'database5761',

password: 'amberdog',

port: 5432,

ssl: {

rejectUnauthorized: false}

})

Note that we have specified that ssl is set to ‘false’! This is because, by default, pg will try to use encryption. However, our database has been setup without encryption.

**UNABLE TO CONNECT WITHOUT SSL SET TO FALSE**

This was a problem that was encountered when trying to connect to the PSQL database. This is because SSL has been set to on by default.

A screenshot of a computer

Description automatically generated

Notice that the DB instance parameter group has a setting. In this window it has been set to ‘noname’. This is a parameter group that has been created retrospectively. The default will be ‘default’.

**A screenshot of a computer

Description automatically generated**

Here is the default Parameter Groups and we can see that a parameter group has been setup for postgres16, which, at time of writing, is the latests PSQL engine.

By clicking into parameter group, we can see that there are many different settings:

A screenshot of a computer

Description automatically generated

**A screenshot of a computer

Description automatically generated**Most notably one of the values is ‘rds.force\_ssl’ which is set to 1. This parameter determines whether SSL (Secure Sockets Layer) is enforced for connections to your RDS instance. When this parameter is set to 1, it means that all connections to the RDS instance must use SSL encryption. SSL encryption ensures that data transmitted between the client application and the database instance is encrypted, providing an additional layer of security, especially important when sensitive data is involved.

By setting this option to 0 (false) it allows connections to the RDS instance without SSL encryption. This is generally not recommended for production environments where security is a concern.

For now we can create our own custom parameter group with rds.force\_ssl set to 0.

**Create parameter Group**

A screenshot of a computer

Description automatically generated

Click on create new parameter group and create the following settings. Finally click create.

**A screenshot of a computer

Description automatically generated**

Notice that our new parameter group has been created.

Click on edit:

A screenshot of a computer

Description automatically generated

Locate the value and set it to 0. Then save changes:

**A screenshot of a computer

Description automatically generated**

Now we can go back to the database and click on Modify:

**A screenshot of a computer

Description automatically generated**

Now attach our newly created parameter group:

**A screenshot of a computer

Description automatically generated**

**Populating PSQL form JSON**

This code uses JavaScript to create a new database instance called ‘temp\_staff’. Temp\_staff will be used to store data from a JSON backup. It will simulate setting up a database for the first time.

The code will read a JSON backup file, before re-populating the database. Here is the final code.

async *function* createTables() {

try {

await client.connect(); // Connect to the database

console.log('< Connected to database! >');

// Reads SQL Queries in '02-create-tables.sql'

*const* sqlQueries = fs.readFileSync('db/02-create\_tables.sql', 'utf8').split(';')

// Executes SQL Queries to create tables.

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

console.log(`Query number; ${i}.`)

await client.query(sqlQueries[i])

}

//

*const* data = fs.readFileSync('db/2024-03-05 15-51-00.730403.json', 'utf8')

*const* jsonData = JSON.parse(data);

for (*const* key in jsonData) {

// Each key represents individual tables.

// rows are the names of the rows.

*const* rows = *Object*.keys(jsonData[key][0]).join(', ');

// Generate the values by mapping over data.

*const* values = jsonData[key].map(*x* *=>* *Object*.values(*x*));

// Create query.

*const* query = format(`INSERT INTO ${key} (${rows}) VALUES %L;`, values);

// Insert query. Use await to wait

// for query to finish executing before moving on.

await client.query(query);

console.log('Added to', key);

}

} catch (error) {

console.error('Error:', error);

} finally {

await client.end(); // Close the client connection.

}

}

createTables(); // Call the async function to execute the queries.

Code Breakdown

Manage Connection

*const* client = new Client ({

user: 'postgres',

host: 'database-5761i.cfk2gikqsjhw.eu-west-2.rds.amazonaws.com',

database: 'database5761',

password: 'amberdog',

port: 5432,

ssl: {

rejectUnauthorized: false}

})

This code hardcode the necessary information to connect to our PSQL database.

A screenshot of a computer screen

Description automatically generatedFirstly, we have a separate SQL file. In this sql file we can see that there are multiple SQL queries. By keeping the files separate, we can edit the SQL file whilst maintain easier to read code in our JavaScript file. In order to read the information we need to use fs.readFileSync. Note that fs.readFileSync is a synchronous operation, which means it will block the execution of the code until it completes reading the file. If the SQL file is large, we want to ensure that it has been fully read, before continuing.

*const* sqlQueries = fs.readFileSync('db/02-create\_tables.sql', 'utf8').split(';')

Because each query ends with a ‘;’ we can use the .split(‘;’) function to create an array, where each element is created by splitting the string at every occurrence of the semicolon. Each element will become each individual query.

Now that we have multiple queries saved, we can begin to iterate through them, and add them to the database.

// Executes SQL Queries to create tables.

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

console.log(`Query number; ${i}.`)

await client.query(sqlQueries[i])

}

Notice that we use the await key word to make sure that the query finishes before continuing.

**Adding Data**

Now that we have finished setting up the schema of the database, we can think about adding data to it!

*const* data = fs.readFileSync('db/2024-03-05 15-51-00.730403.json', 'utf8')

*const* jsonData = JSON.parse(data);

First of all we need to read a json file that is stored locally, where all of the database information has been stored. We first read the file which creates a large string – and then parse it, which creates a useable object with key value pairs.

A screenshot of a computer

Description automatically generated

Now we have a useable object we can begin to iterate through it to build up our queries. In the above example we can see that one of the keys will be ‘counterparty’. The value will be ALL of the data in the counterparty table.

for (*const* key in jsonData) {

// Each key represents individual tables.

// rows are the names of the rows.

*const* rows = *Object*.keys(jsonData[key][0]).join(', ');

Each key represents each table.

If we:

jsonData[key][0]

We get:

A screen shot of a computer

Description automatically generated

This is because we are simply accessing the first piece of data! From this we can use Object.keys() is a built in JavaScript method that returns an array of a given object’s enumerable property names. In this instance it would return; [counterparty\_id, counterparty\_legal\_name…] and so on. We use the ‘.join(, )’ method to turn the array into a string. This would create a string like this; “counterparty\_id, counterparty\_legal\_name…” this correctly edits the string into a format we can use to make a query!

Next we need to extract the values:

// Generate the values by mapping over data.

*const* values = jsonData[key].map(*x* *=>* *Object*.values(*x*));

Finally we can create our query:

*const* query = format(`INSERT INTO ${key} (${rows}) VALUES %L;`, values);

This query can then be used to insert data into our psql database.

await client.query(query);

The await query is used to ensure that each query finishes executing before moving on!

} catch (error) {

console.error('Error:', error);

} finally {

await client.end(); // Close the client connection.

}

We use catch to capture any errors that occur.

Finally is used to close the database connection when it has finished executing.

**Problems**

**Auto Incrementation**

<https://www.postgresql.org/docs/10/functions-sequence.html>

Here is the query for our staff table:

DROP TABLE IF EXISTS staff;

CREATE TABLE staff (

staff\_id *SERIAL* PRIMARY KEY,

first\_name *VARCHAR*,

last\_name *VARCHAR*,

department\_id *INT*,

email\_address *VARCHAR*,

last\_updated *DATE*,

created\_at *DATE*

);

We can see that staff\_id is a serial primary key. This means that postgres will automatically increment the value when you add a new value to the staff table.

Initially when we are seeding the data from our staff table we are reading from a json file.

A screenshot of a computer

Description automatically generated

The json file is simulating the backend of a shop or a database system. This means that we want to preserve the data in a 1:1 relationship. We want the staff\_id to be the same in our database. When we are seeding our database for the first time, we are going to implicitly define the staff\_id field.

However this could cause a problem. Lets say we implement 20 values. If we try to add a new value WITHOUT adding the staff\_id field, we would normally expect PSQL to automatically update the staff\_id value with the value 21.

Because we have added data implicitly. PSQL has not incremented a background value that keeps track of the current value. We therefore need to set the value ourselves!

if(key == 'staff'){

console.log(rows)

console.log(values)

await client.query(`SELECT setval ('staff\_staff\_id\_seq', 20);`)

}

}

Now when we add new data, it should automatically update correctly. The next value will be 21!