Module 12
RDS Database

Overview

Amazon's cloud relational databases offering is known as **RDS** (Relational Database Services). RDS does not include Dynamo DB, which is not a relational database.

RDS databases are **managed services**. By "managed" it means that Amazon does the heavy lifting of doing all administrative operations (backup, restore, etc.). It also means that the customer cannot remote connect to the cloud machine/VM where the database is running. In most cases it is a **multi-tenant** environment where a server is being used to host other customers' databases as well (a customer can also request **single-tenancy**, which is a bit more costly). Therefore, from an installation perspective, this is different from owning a local database where you have access to the server where it is running. If a customer desires to have this capability, one option is to start an EC2 machine and install a database server software on it. This would be different because the customer is managing the server and the database.

Note that managed services does not mean guaranteeing that a database schema is well-designed or includes things like tuning a database to say query data faster. This is still the customer's responsibility. Therefore, if you create a table that becomes extremely large but you forget to create an index for it, AWS does not detect this and remind you to do so to improve search speed. The design of the database schema and all other objects needed (e.g., indexes, etc.) is still the responsibility of the customer.

In this module we will create a PostgreSQL database, create a table in it, and write to the table from a Lambda function.

A brief history of PostgreSQL

PostgreSQL was developed at University of California, Berkley in the mid-1980s by Michael Stonebraker who won the Turing Award in 2014 for his research on relational databases. It is now an open-source database maintained by the *PostgreSQL Global Development Group*. Its official website is https://www.postgresql.org/.

Create a PostgreSQL Database

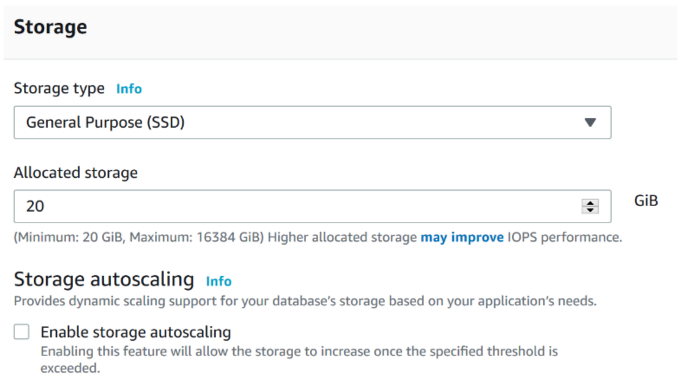
1. Go to Canvas, start the lab, and navigate to the AWS Management Console. Then go to the RDS service.
2. From the navigation bar on the left, click **Databases**.
3. Click the **Create database** button.
4. Leave the **Standard create** option selected. From Engine options, choose the **PostgreSQL** one.

5. Don't change the version. Take the suggested one.
6. From Templates, choose the **Free tier** option.
7. In the Settings section, use the following:

DB instance identifier, enter:	mod12pginstance
Master username, enter:	postgres
Master password, enter:	cs455pass

8. In the DB instance class section, take the suggested default: db.t3.micro.

9. Fill the Storage section as shown below (do not enable storage auto scaling since we are only inserting a few rows to a table – so uncheck the **Enable storage autoscaling** checkbox):



Storage

Storage type Info

General Purpose (SSD)

Allocated storage

20 GiB

(Minimum: 20 GiB, Maximum: 16384 GiB) Higher allocated storage may improve IOPS performance.

Storage autoscaling Info

Provides dynamic scaling support for your database's storage based on your application's needs.

☐ Enable storage autoscaling

Enabling this feature will allow the storage to increase once the specified threshold is exceeded.

10. In the Connectivity section, set **Public access** to **Yes**.

11. In the Database authentication section, take the default **Password authentication** option.

12. Expand the **Additional configuration** section.

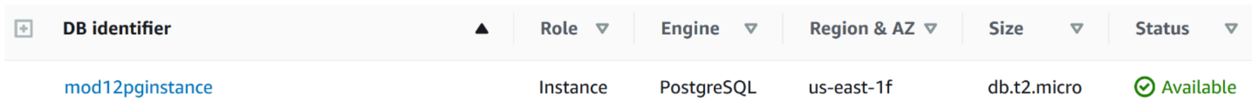
    Initial database name: SalesDB
    Uncheck the **Enable automatic backups**
    Uncheck the **Enable auto minor version upgrade**

    Leave everything else in this section as-is.

13. In the Maintenance section, uncheck the **Enable auto minor version upgrade** checkbox.

14. Click the **Create database** button.

15. Go back to the RDS page. You should see **mod12pginstance** in the table. After a few minutes, its status should say Available:

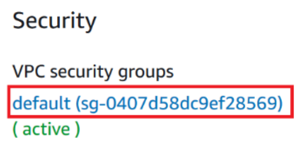


DB identifier	Role	Engine	Region & AZ	Size	Status
mod12pginstance	Instance	PostgreSQL	us-east-1f	db.t2.micro	Available

Setting values in the table should become familiar to you as we progress with the course:

- **t2.micro** is an instance type. It is basically a name for a hardware configuration (CPU, memory, etc.). See what t2.micro has here (click the T2 tab).
- The database was created in AWS region us-east-1 (N. Virginia).
- In us-east-1, in my example it was created in Availability Zone F (remember: us-east-1 has 6 AZs – a to f).
- The Engine is PostgreSQL (RDS offers other databases such as Oracle, SQL Server, MySQL, etc.). We chose the PostgreSQL one.

16. Click the mod12pginstance link. In the Networking section and Security section, take note of the **VPC security groups** name (you will need it later).



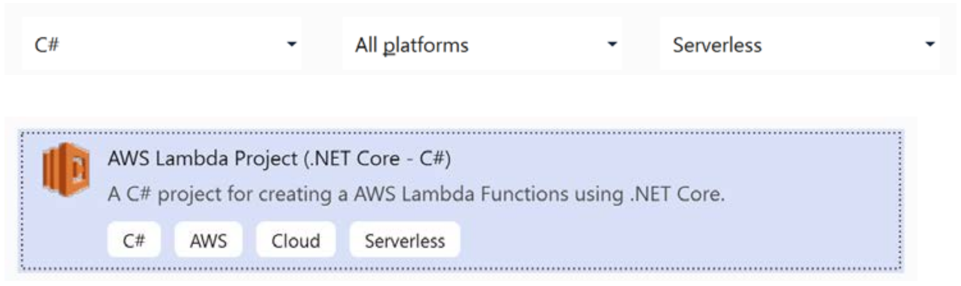
Security

VPC security groups
default (sg-0407d58dc9ef28569)
(active)

Create a Lambda Function

We will create a Lambda function from where we will access the database using a library known as **Npgsql**.

1. Create a Visual Studio solution with name RDSPostgreSQL.
2. In this solution, create a project of type AWS Lambda Project (.NET Core – C#).
   Name the project: PostgreSQLFunction



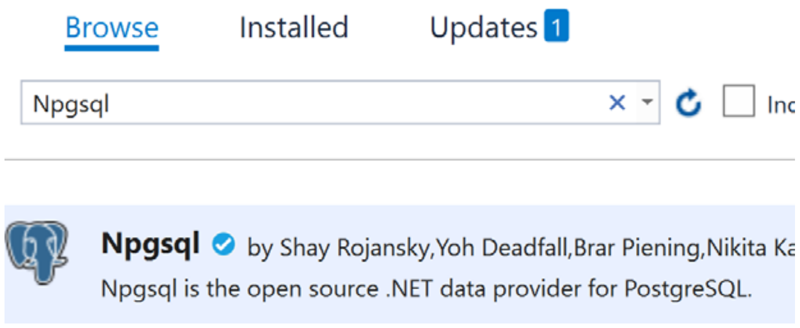
C# | All platforms | Serverless

AWS Lambda Project (.NET Core - C#)
A C# project for creating a AWS Lambda Functions using .NET Core.
C# AWS Cloud Serverless

3. From the blueprints, choose the **Empty Function** template. Then click **Finish**.

4. In Solution Explorer, right-click on **Packages** and choose menu **Manage NuGet Packages…**

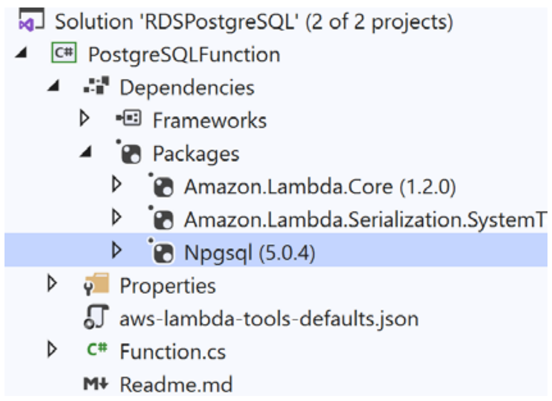
5. Make sure **Browse** is selected. Search for **Npgsql**. Then click the **Install** button on the right.



Npgsql by Shay Rojansky,Yoh Deadfall,Brar Piening,Nikita Ka
Npgsql is the open source .NET data provider for PostgreSQL.

Npgsql is a data provider for PostgreSQL. It has classes that allow you to connect to and interact with a PostgreSQL database.

Your project structure should now look like the below (your Npgsql version might be different than mine. That's OK):



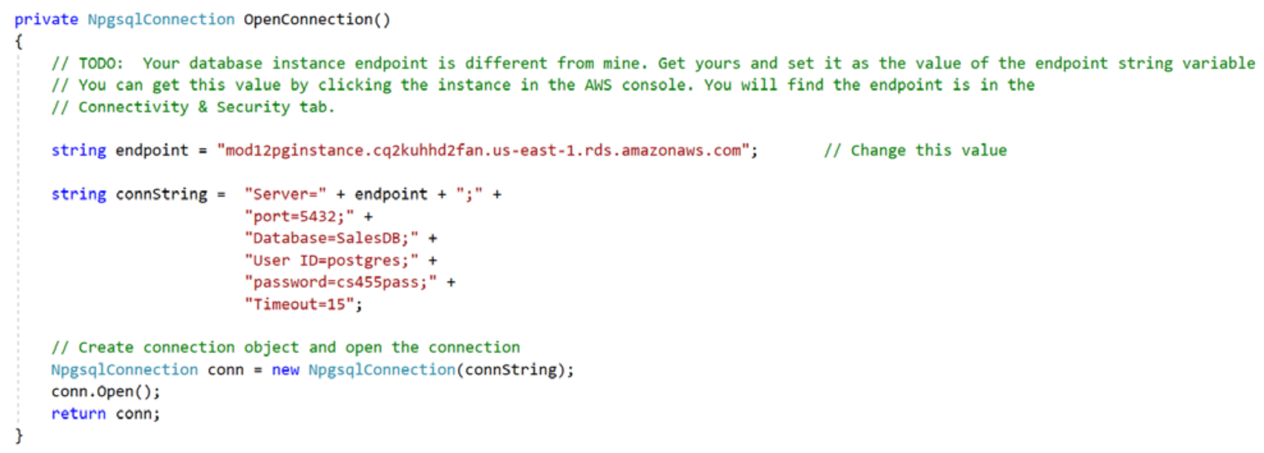
```
Solution 'RDSPostgreSQL' (2 of 2 projects)
◢ PostgreSQLFunction
   ◢ Dependencies
      ▷ Frameworks
      ◢ Packages
         ▷ Amazon.Lambda.Core (1.2.0)
         ▷ Amazon.Lambda.Serialization.SystemT
         ▷ Npgsql (5.0.4)
   ▷ Properties
     aws-lambda-tools-defaults.json
   ▷ Function.cs
     Readme.md
```

6. Add the following code to the Function class. In this code we simply test that we can successfully open a connection to the PostgreSQL database.

At the top of Function.cs, add these using statements:

```
using Npgsql;
using System.Data;
```

Add this function to the `Function` class. Note that your endpoint value is different. Update it based on your database instance settings (note that concatenating lots of strings using the + operator is not recommended – we are doing it here to keep things simple):



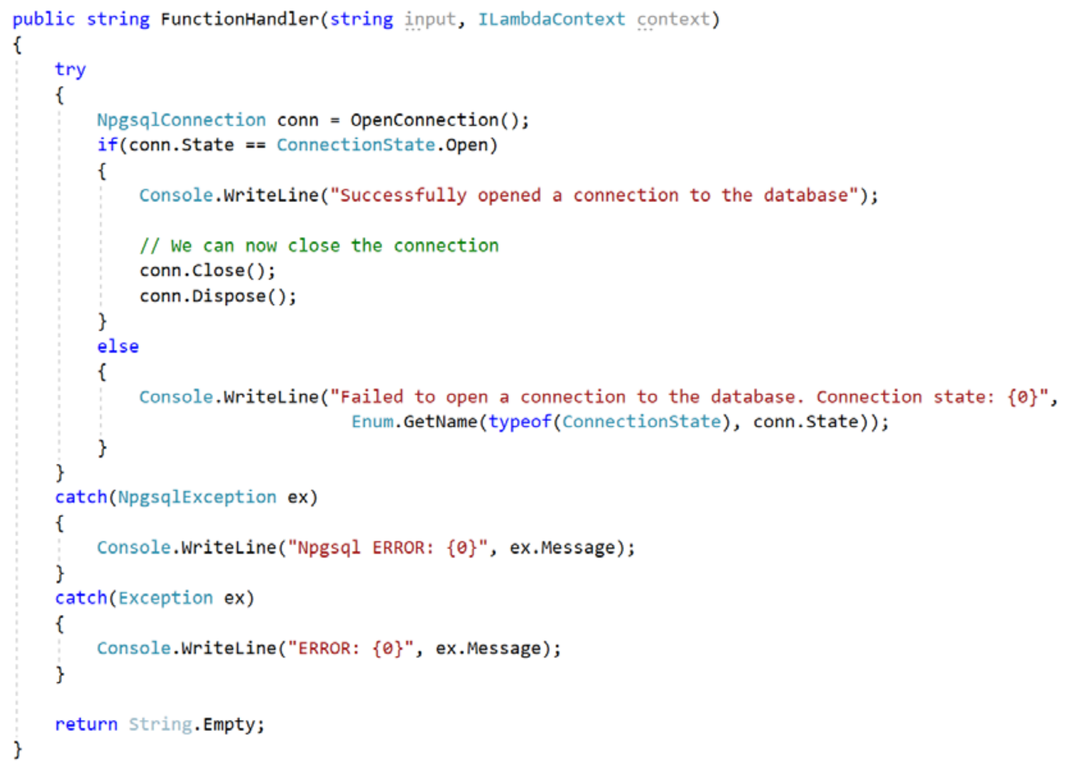
```
private NpgsqlConnection OpenConnection()
{
    // TODO:  Your database instance endpoint is different from mine. Get yours and set it as the value of the endpoint string variable
    // You can get this value by clicking the instance in the AWS console. You will find the endpoint is in the
    // Connectivity & Security tab.

    string endpoint = "mod12pginstance.cq2kuhhd2fan.us-east-1.rds.amazonaws.com";       // Change this value

    string connString =  "Server=" + endpoint + ";" +
                         "port=5432;" +
                         "Database=SalesDB;" +
                         "User ID=postgres;" +
                         "password=cs455pass;" +
                         "Timeout=15";

    // Create connection object and open the connection
    NpgsqlConnection conn = new NpgsqlConnection(connString);
    conn.Open();
    return conn;
}
```

Replace the code of `FunctionHandler` with this:



```
public string FunctionHandler(string input, ILambdaContext context)
{
    try
    {
        NpgsqlConnection conn = OpenConnection();
        if(conn.State == ConnectionState.Open)
        {
            Console.WriteLine("Successfully opened a connection to the database");

            // We can now close the connection
            conn.Close();
            conn.Dispose();
        }
        else
        {
            Console.WriteLine("Failed to open a connection to the database. Connection state: {0}",
                              Enum.GetName(typeof(ConnectionState), conn.State));
        }
    }
    catch(NpgsqlException ex)
    {
        Console.WriteLine("Npgsql ERROR: {0}", ex.Message);
    }
    catch(Exception ex)
    {
        Console.WriteLine("ERROR: {0}", ex.Message);
    }

    return String.Empty;
}
```

7. Build the solution and verify that there are no errors.

8. Publish to AWS:

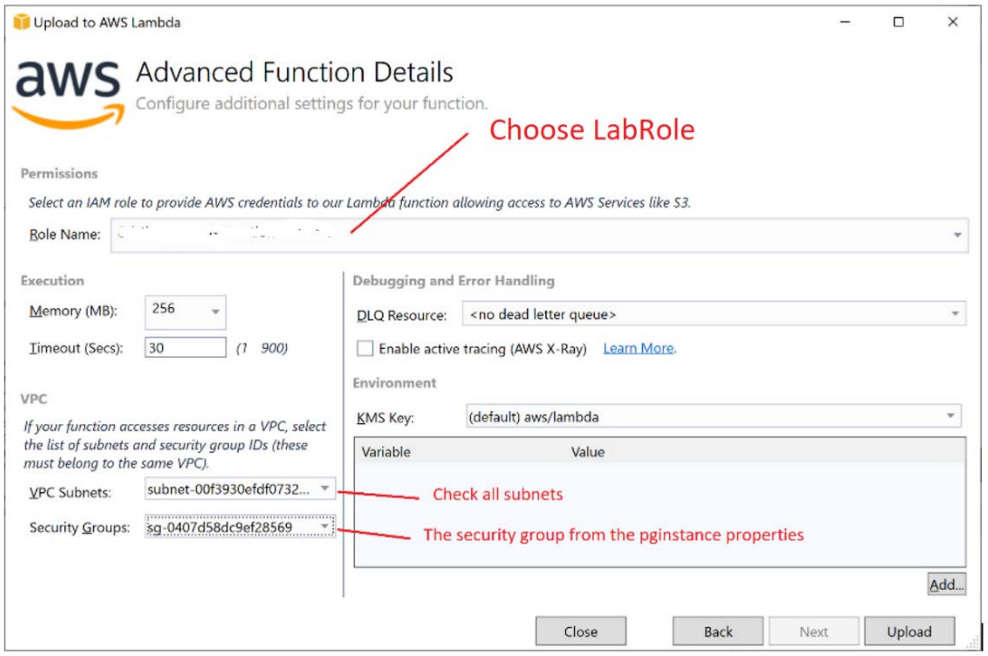
   NOTE: Renew your credentials now if they are more than 3 hours old.

   a. Right-click project **PostgreSQLFunction** and choose **Publish to AWS Lambda…**.

b. In the publish wizard, choose the US East (N. Virginia) region. For Function Name enter: **postgresql-function**

c. Click **Next**.

d. Fill in the Advanced Function Details screen as shown below:


Upload to AWS Lambda

Advanced Function Details

Configure additional settings for your function.

Choose LabRole

Permissions

Select an IAM role to provide AWS credentials to our Lambda function allowing access to AWS Services like S3.

Role Name: [redacted]

Execution

Memory (MB): 256

Timeout (Secs): 30 (1 900)

VPC

If your function accesses resources in a VPC, select the list of subnets and security group IDs (these must belong to the same VPC).

VPC Subnets: subnet-00f3930efdf0732...

Security Groups: sg-0407d58dc9ef28569

Debugging and Error Handling

DLQ Resource: <no dead letter queue>

☐ Enable active tracing (AWS X-Ray) Learn More.

Environment

KMS Key: (default) aws/lambda

Variable	Value

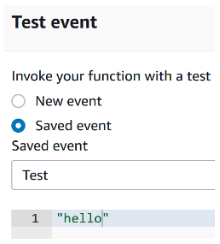
Check all subnets

The security group from the pginstance properties

e. Click the **Upload** button.

Test that your Lambda Function Can Connect to PostgreSQL

1. Go to the AWS Console → Lambda service. Click the **postgresql-function**.
2. Click the **Test** tab.
3. Choose **New event** option.
4. In the Name textbox, enter: Test
5. Click the **Save changes** button.
6. Now while **Saved event** / **Test** are selected, change the input to some string (e.g., "hello").


Test event

Invoke your function with a test

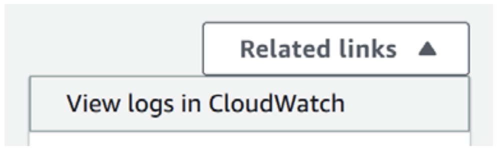
- [ ] New event
- [✓] Saved event

Saved event

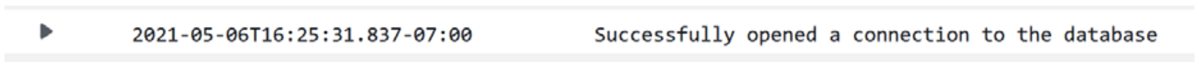
Test

```
"hello"
```

7. Click the **Test** orange button.
8. If the test succeeded, click the **Monitor** tab. Then click **Related links → View logs in CloudWatch**

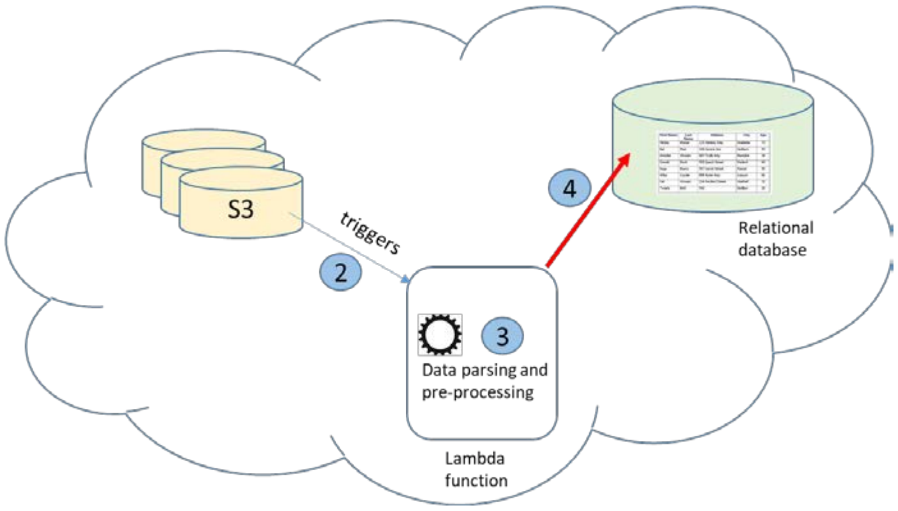


9. If you were able to successfully connect to the database you should see the message you printed to the console:



```
▶      2021-05-06T16:25:31.837-07:00          Successfully opened a connection to the database
```

10. Once you connect to the database, you can do all CRUD operations (select, insert, update, delete). What we have demonstrated is step 4 in the picture below (Ignore S3 in the picture below). That is, connect from Lambda to the PostgreSQL database.



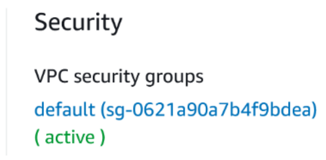
Working with the Database from a Local Client GUI

Sometimes it is useful to look at a database using a GUI application. We will do that in this section.

There are many GUIs that connect to most databases. You are welcome to use anyone you are familiar with. I will use a popular one for PostgreSQL called pgAdmin.

1. Go to RDS service in the AWS console and click your DB instance.

2. In the **Connectivity and security** tab, click the VPC security group link.

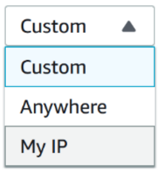


Security

VPC security groups

default (sg-0621a90a7b4f9bdea)
(active)

3. Click button **Action** – **Edit inbound rules**.
4. Click the **Add rule** button. This creates a second rule row.
5. In the new rule row that gets created, open the Source dropdown and select My IP:

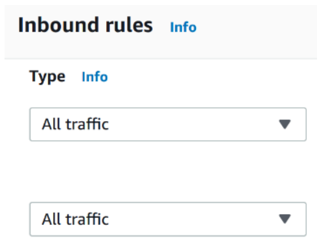


This automatically retrieves your computer's IP address.

6. Change the Type dropdown to "All traffic"

7. Click the **Save rules** button.

Your Inbound rules table should now have 2 rows: one row with IP 0.0.0.0/0 and another with your local computer IP address.



Inbound rules Info

Type Info

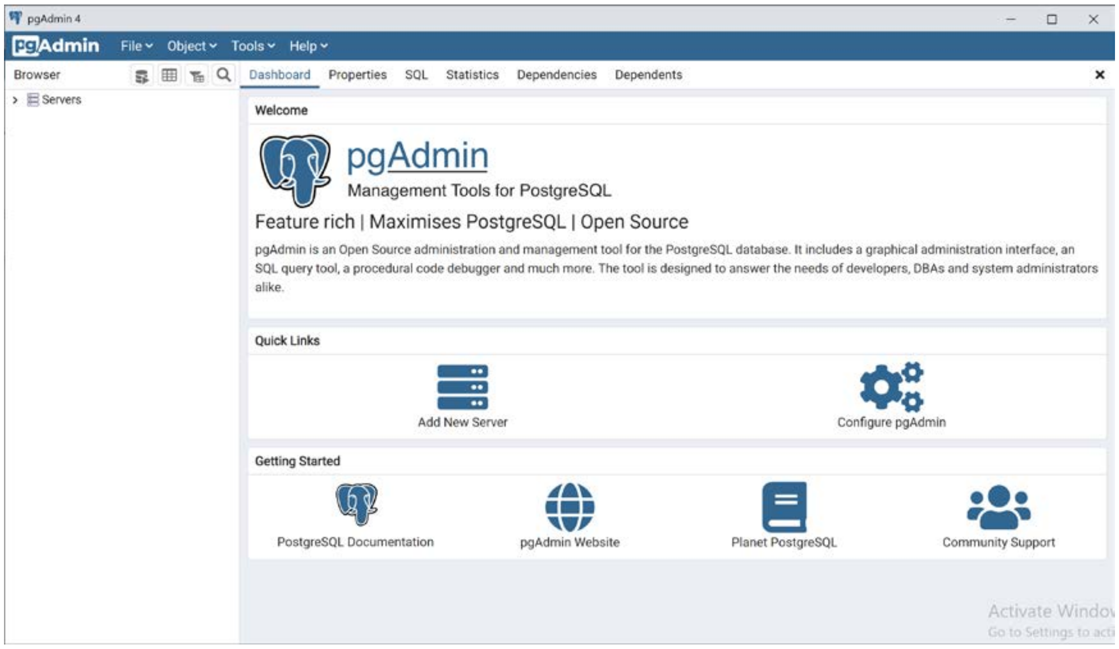
All traffic

All traffic

8. Go to the pgAdmin site and download and install it. Make sure you choose the one relevant to your OS.

9. Run the installation wizard taking all default settings.

10. You should now be able to open it: click the Windows button and start typing: pgAdmin.



pgAdmin

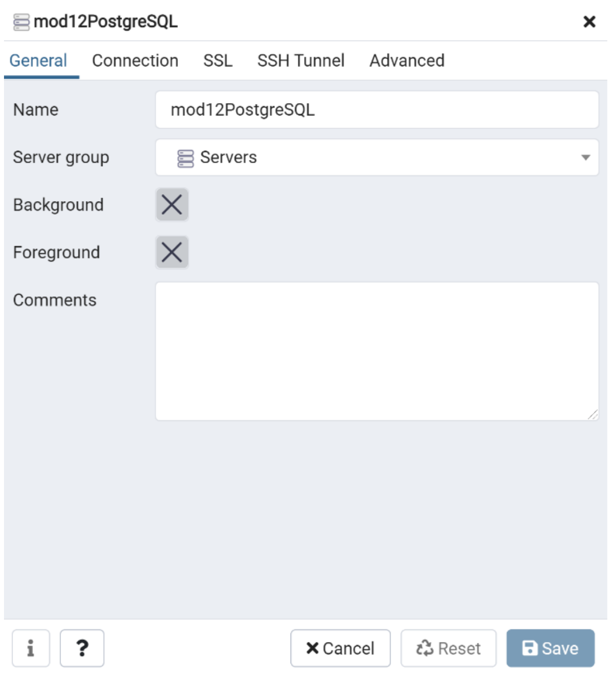
Management Tools for PostgreSQL

Feature rich | Maximises PostgreSQL | Open Source

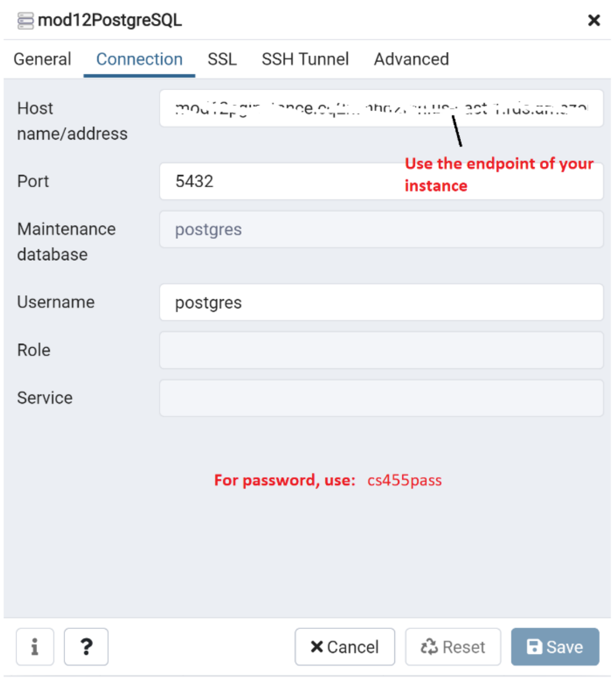
pgAdmin is an Open Source administration and management tool for the PostgreSQL database. It includes a graphical administration interface, an SQL query tool, a procedural code debugger and much more. The tool is designed to answer the needs of developers, DBAs and system administrators alike.

11. Follow the instructions in this AWS documentation page (section "Using pgAdmin to connect to a PostgreSQL DB instance").

My connection settings look like this:



Name	mod12PostgreSQL
Server group	Servers
Background	
Foreground	
Comments	

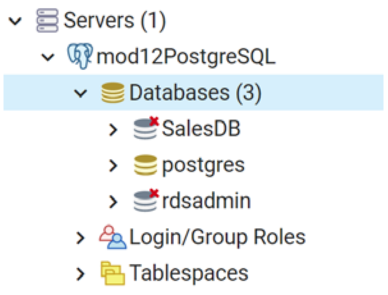


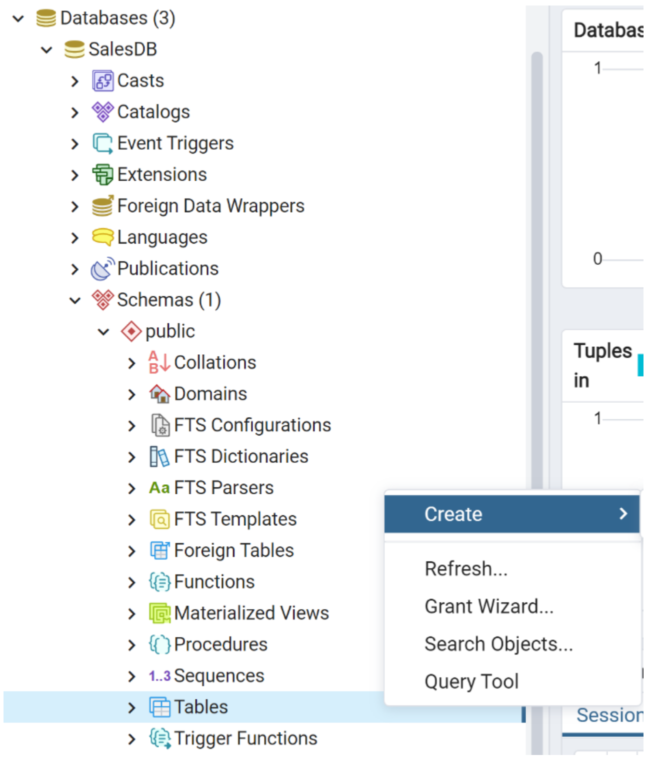
Host name/address	[illegible]
Port	5432
Maintenance database	postgres
Username	postgres
Role	
Service	

Use the endpoint of your instance

For password, use: cs455pass

12. I can now see my RDS PostgreSQL database and interact with it (add tables, etc).





Exercise to do on your own

1. Use pgAdmin to create a table on the SalesDB (e.g., Products table)
2. Add a few columns to the Products table (e.g., ProductID, ProductName).
3. Go back to your Lambda function and practice entering rows into the Product table (The Npgsql namespace should have classes you can use to insert data – look at the documentation).

What to Submit

Nothing to submit for this module.