# 1. Create Azure Database for PostgreSQL

### 1.1 Via Azure Portal

1. Go to the [**Azure Portal**](https://portal.azure.com/) and click Create a resource.
2. Search for “Azure Database for PostgreSQL flexible server” (the current recommended option).
3. Configure Server:
   * Subscription / Resource Group: pick existing or create new.
   * Server Name: a unique name (e.g., my-pgserver-123).
   * Region: pick a region near you.
   * Workload Type: Typically “General purpose” or “Burstable” for dev/test.
   * Admin Username/Password: e.g., pgadmin / MySecurePassword123
   * Version: Choose a supported PostgreSQL version (e.g., 14).
4. Networking:
   * You can choose “Public Access (allowed IP addresses).”
   * Add your local IP to the firewall rules if needed.
5. Review + Create to finalize.

Testing Step: Once created, use a client like [Azure Data Studio](https://learn.microsoft.com/sql/azure-data-studio) or psql to confirm you can connect:

bash

CopyEdit

psql "host=my-pgserver-123.postgres.database.azure.com port=5432 dbname=postgres user=pgadmin password=MySecurePassword123 sslmode=require"

If you can connect, you’re good to go.

**Create postgre sql schema , and insert data**

**create function app**

# **1. Set Up the Azure Function App Environment**

## 1.1 Install Prerequisites

1. Node.js (LTS or Current)  
   [Download and install Node.js](https://nodejs.org/) if you don’t have it already.
2. Azure Functions Core Tools  
   This lets you develop and run Azure Functions locally.
   * On most systems, you can install via npm:

bash

CopyEdit

npm install -g azure-functions-core-tools@4

* + See [official installation docs](https://learn.microsoft.com/azure/azure-functions/functions-run-local) for other options.

1. PostgreSQL Database
   * If you plan to use Azure Database for PostgreSQL, create a new server (e.g., via [Azure Portal](https://portal.azure.com/)) or connect to an existing PostgreSQL instance.
   * Take note of the connection details (host, port, username, password, database).

## 1.2 Create a New Azure Functions Project

1. Initialize the Project Folder  
   Create a new folder (e.g., pg-azure-functions), then initialize a Functions project in JavaScript (Node.js):

bash

CopyEdit

mkdir pg-azure-functions

cd pg-azure-functions

func init . --javascript

1. Add the pg Library (for PostgreSQL)  
   Install [node-postgres](https://www.npmjs.com/package/pg):

bash

CopyEdit

npm install pg

1. (Optional) Add GraphQL Libraries  
   For a GraphQL endpoint, we’ll use [apollo-server-azure-functions](https://www.npmjs.com/package/apollo-server-azure-functions) and graphql:

bash

CopyEdit

npm install apollo-server-azure-functions graphql

# 2. Create a REST Endpoint to Query PostgreSQL

We’ll create a new Azure Function (GetItems) that handles HTTP requests and queries the database.

## 2.1 Generate a Function

From your project folder:

bash

CopyEdit

func new --name GetItems --template "HTTP trigger"

This creates a new folder GetItems with index.js. By default, it looks like:

js

CopyEdit

module.exports = async function (context, req) {

context.log('JavaScript HTTP trigger function processed a request.');

const name = (req.query.name || (req.body && req.body.name));

const responseMessage = name

? "Hello, " + name

: "This HTTP triggered function executed successfully. Pass a name in the query string or in the request body for a personalized response.";

context.res = {

// status: 200, /\* Defaults to 200 \*/

body: responseMessage

};

}

## 2.2 Connect to PostgreSQL in the Function

Replace the contents of GetItems/index.js with the following:

js

CopyEdit

const { Client } = require("pg");

/\*\*

\* Azure Function: GetItems

\* This function uses a GET request to retrieve rows from a 'items' table in PostgreSQL.

\*/

module.exports = async function (context, req) {

// Log an informational message

context.log("GetItems function processing request...");

// DB connection details - prefer environment variables for secrets!

const config = {

user: process.env.DB\_USER, // e.g. "pgadmin"

password: process.env.DB\_PASSWORD, // e.g. "MySecurePassword123"

host: process.env.DB\_HOST, // e.g. "my-postgres-server.postgres.database.azure.com"

database: process.env.DB\_DATABASE, // e.g. "mydbname"

port: Number(process.env.DB\_PORT || 5432),

ssl: { rejectUnauthorized: false } // For Azure SSL

};

// Create a new client

const client = new Client(config);

try {

// Connect to PostgreSQL

await client.connect();

// Basic SELECT from an "items" table

const result = await client.query(`SELECT id, name FROM items LIMIT 50;`);

// Close the client

await client.end();

// Return the rows in JSON format

context.res = {

status: 200,

body: result.rows,

headers: { "Content-Type": "application/json" },

};

} catch (err) {

context.log.error("Error connecting to PostgreSQL:", err);

// Return 500 on error

context.res = {

status: 500,

body: { error: err.message },

};

}

};

### 2.3 Use Environment Variables for Secrets

Create a .env file in the root of your functions project (this will not be committed if your .gitignore is standard). For example:

plaintext

CopyEdit

DB\_USER=pgadmin

DB\_PASSWORD=MySecurePassword123

DB\_HOST=my-postgres-server.postgres.database.azure.com

DB\_DATABASE=mydbname

DB\_PORT=5432

Azure Functions Core Tools will automatically load these into process.env when running locally.

## 2.4 Test Locally

1. Start the local Azure Functions runtime:

bash

CopyEdit

func start

1. Call the REST Endpoint:

bash

CopyEdit

curl http://localhost:7071/api/GetItems

* + If all goes well, you should see a JSON array of items from your PostgreSQL table.

Incremental Test: If you see an error, check the function logs in the terminal to confirm the database connection details are correct.

# **4. Deploy to Azure**

Once your REST and GraphQL endpoints work locally, you can deploy them to an Azure Function App.

## 4.1 Create an Azure Function App (Portal or CLI)

Using the Azure CLI:

bash

CopyEdit

# Log into Azure

az login

# Create a resource group if you don't have one

az group create --name MyResourceGroup --location eastus

# Create a new Storage Account (required by Azure Functions)

az storage account create \

--name mystorageaccount123 \

--location eastus \

--resource-group MyResourceGroup \

--sku Standard\_LRS

# Create the Function App

az functionapp create \

--resource-group MyResourceGroup \

--consumption-plan-location eastus \

--runtime node \

--functions-version 4 \

--name my-pg-functions \

--storage-account mystorageaccount123

## 4.2 Configure Application Settings (Environment Variables)

Your connection credentials can be stored as Application Settings in the Function App:

bash

CopyEdit

az functionapp config appsettings set \

--name my-pg-functions \

--resource-group MyResourceGroup \

--settings DB\_USER=pgadmin \

DB\_PASSWORD=MySecurePassword123 \

DB\_HOST=my-postgres-server.postgres.database.azure.com \

DB\_DATABASE=mydbname \

DB\_PORT=5432

## 4.3 Deploy Your Code

In your local project:

bash

CopyEdit

func azure functionapp publish my-pg-functions

This bundles your code and uploads it to the specified Function App.

Testing:

* Once published, you can visit:

arduino

CopyEdit

https://my-pg-functions.azurewebsites.net/api/GetItems

for the REST endpoint, and

arduino

CopyEdit

https://my-pg-functions.azurewebsites.net/api/GraphQL

for the GraphQL endpoint.

* Use curl or a browser to ensure the endpoints respond correctly with data from your Azure PostgreSQL database.

# **5. Final Verification and Notes**

1. Check Logs in Azure Portal  
   Go to the Function App -> Monitor or use az functionapp log tail --name my-pg-functions --resource-group MyResourceGroup to see real-time logs.
2. Secure Endpoints
   * Out of the box, these endpoints are publicly accessible.
   * You can add [Function-level auth](https://learn.microsoft.com/azure/azure-functions/functions-bindings-http-webhook-trigger?tabs=csharp" \l "authorization-levels), [JWT tokens](https://learn.microsoft.com/azure/azure-functions/functions-security-authentication), or [Azure AD integration](https://learn.microsoft.com/azure/active-directory/develop/) as needed.
3. Performance Considerations
   * Each function invocation connects to the PostgreSQL database. For high load, consider using connection pooling (e.g., pg-pool) or a dedicated connection approach.
   * Keep in mind the cold start behavior of Azure Functions in the Consumption plan.

# **Summary**

* Initialize a Node.js Azure Functions project (func init).
* Add the pg library to connect to PostgreSQL.
* Create separate HTTP-triggered functions for REST (GetItems) and GraphQL (GraphQL).
* Connect to your PostgreSQL database by reading credentials from environment variables.
* Test locally with func start.
* Deploy to Azure with func azure functionapp publish ....
* Verify your endpoints in production.

By following these steps, you’ll have both REST and GraphQL endpoints for your PostgreSQL database running on Azure Functions, giving you flexibility in how your application or clients consume the data.