**Serverless Discord Bot Using AWS Lambda and API Gateway**

**Project Overview**

This project demonstrates how to create a **serverless Discord bot** using **AWS Lambda** and **API Gateway**, leveraging **Discord’s Interactions API and Slash Commands**. The goal is to eliminate the need for persistent servers by triggering lightweight, stateless Lambda functions on-demand when users interact with the bot via commands like /foo.

The project integrates:

* Discord’s Interaction model for privacy and scalability
* AWS serverless compute services for cost-effective, event-driven execution

**Key Features**

* **Serverless architecture**: No always-on server needed. AWS Lambda functions run on demand.
* **Secure request validation**: Uses tweetnacl to verify Discord signatures.
* **Slash command support**: Register and handle custom Discord commands like /foo.
* **Low-latency responses**: Responds to commands within Discord’s required 3-second window.
* **Environment-based configuration**: Secrets like DISCORD\_TOKEN and PUBLIC\_KEY are managed securely.
* **Modular command registration**: Easily extendable command setup using @discordjs/builders.
* **Deployable via AWS CLI or Console**: Offers flexibility for developers.
* **CORS-enabled endpoint**: Open to HTTP requests from Discord API.

**Tech Stack**

| **Component** | **Technology Used** |
| --- | --- |
| **Runtime Environment** | AWS Lambda (Node.js 22.x) |
| **HTTP Handling** | AWS API Gateway (HTTP API) |
| **Programming Language** | JavaScript (ESM, Node.js) |
| **Discord Integration** | Discord Developer Portal, Slash Commands |
| **Security** | tweetnacl (request signature verification) |
| **Command Registration** | Axios, dotenv, @discordjs/builders |
| **Dependency Management** | Npm |
| **Bot Framework** | discord.js (for local testing, optional) |
| **Deployment Tools** | AWS Console, AWS CLI |

**Use Cases**

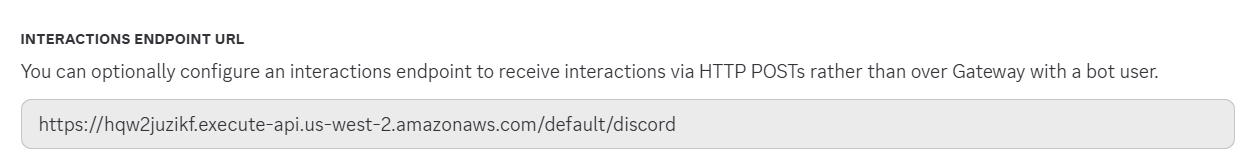
1. **Lightweight Discord Bots**  
   Ideal for bots that respond to slash commands like /help, /info, or /weather without needing real-time message reading.
2. **Education & Demos**  
   Perfect example to demonstrate modern serverless architecture in AWS with a real-world application.
3. **Event-Driven Chat Applications**  
   Bots that react to user actions or events through well-defined API triggers.
4. **Low-Cost Bots for Small Communities**  
   Suitable for servers where continuous uptime is not necessary, optimizing cost-efficiency.
5. **Compliance-Friendly Applications**  
   Fits environments where **privacy** and **limited data processing** are prioritized — no message-reading permission is required.

**Step-by-Step Guid**

Objectives

Recently, Discord has introduced **Interactions** and **Slash Commands** to ease users experience with bots. Partially, the reason was to secure users privacy — bots are now slightly forced to abandon flags for reading messages in Discord Servers — which was previously a point of concern regarding unlawful data processing by bot developers.

These are the most known features, but on top of that, the update has added a possibility to interact with mentioned Interactions through the **Interactions Endpoint URL**.

Developer Dashboard’s Field for Interactions Endpoint URL

This allows us to move from a traditional **listener** app hosted on a server to a **serverless** app , that runs a Lambda Function on each Interaction invocation

* **Connecting AWS Lambda to Discord**

During the coding, we will jump between “AWS”, “Code” and “Discord Developer Portal”.

* **AWS Prerequisites:**

We will use **AWS API Gateway** and **AWS Lambda** for our task. It’s a base toolset for serverless architecture on AWS. Functions will be written in **Node.js**.

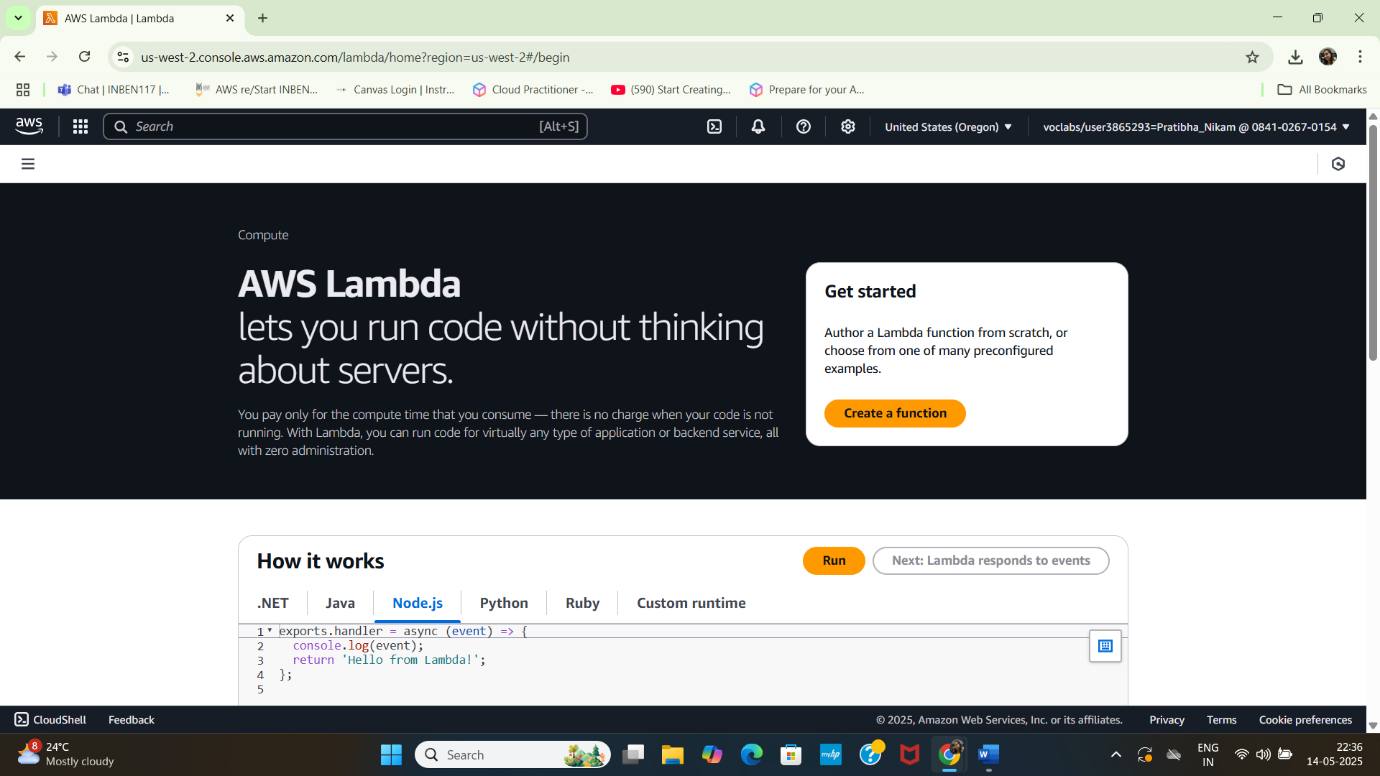
**Part 1: Setup AWS Lambda Function**

**1. Create an AWS Account**

* Visit <https://aws.amazon.com> and sign up.
* Add a payment method (you won't be charged unless you exceed the free tier).

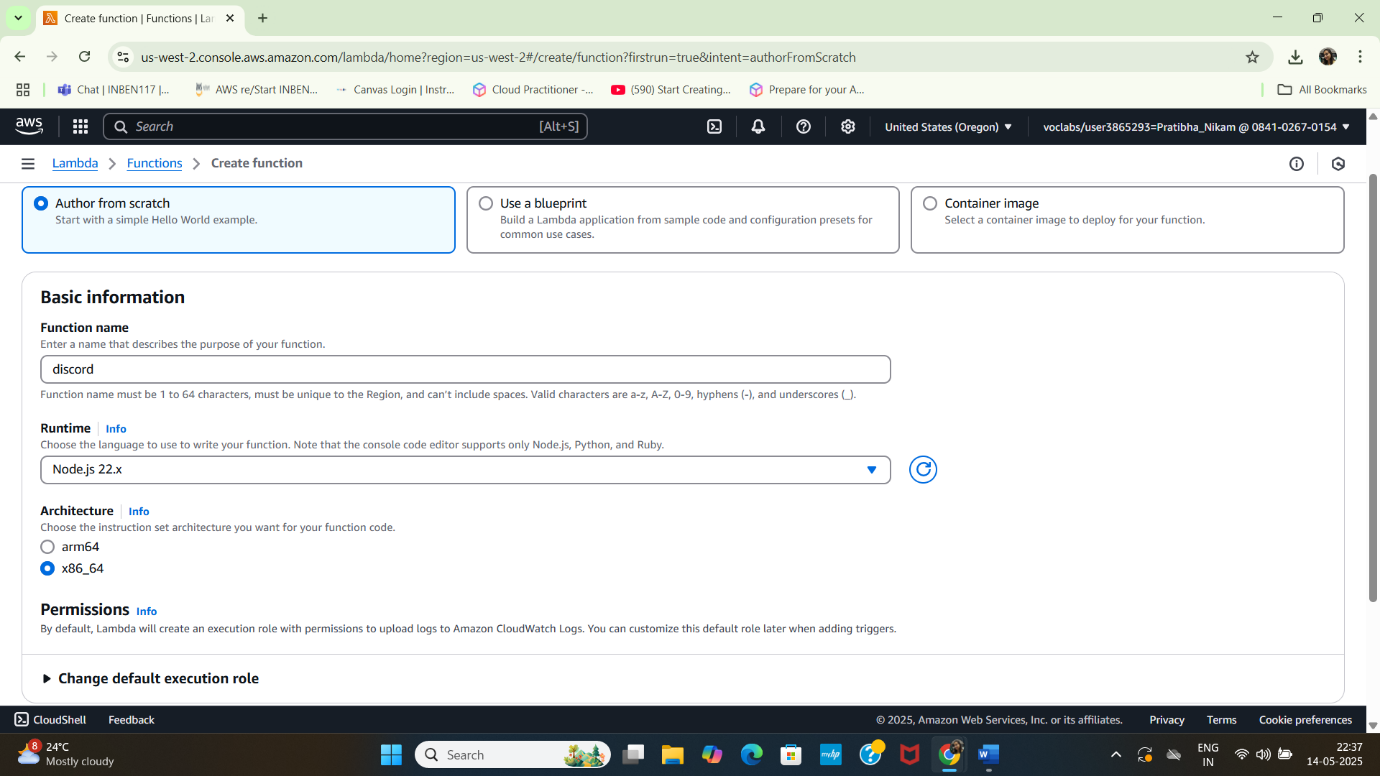
**2. Navigate to Lambda**

* From AWS Console: Use the **search bar** to find Lambda.
* Click **“Functions”** in the left sidebar.
* Click **“Create function”**.



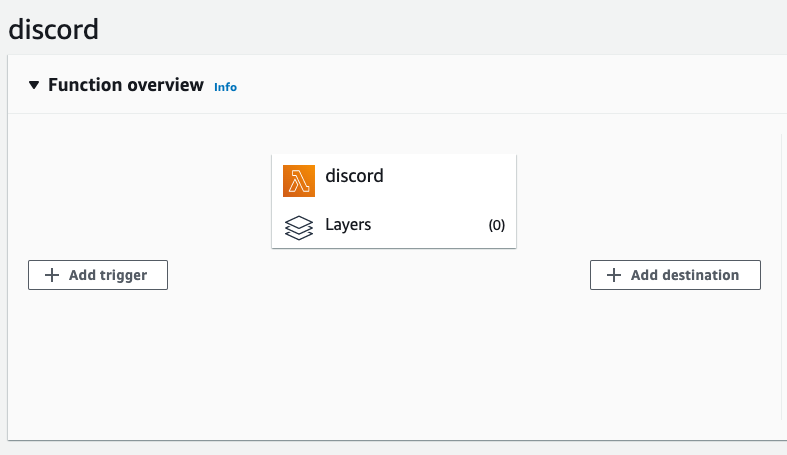
**3. Create a New Function**

* Choose **Author from scratch**.
* Function name: discord
* Runtime: **Node.js 22.x**
* Leave permissions as default (a basic execution role will be created).
* Click **“Create function”**.



Create function panel

After creation, you should see the dashboard below.

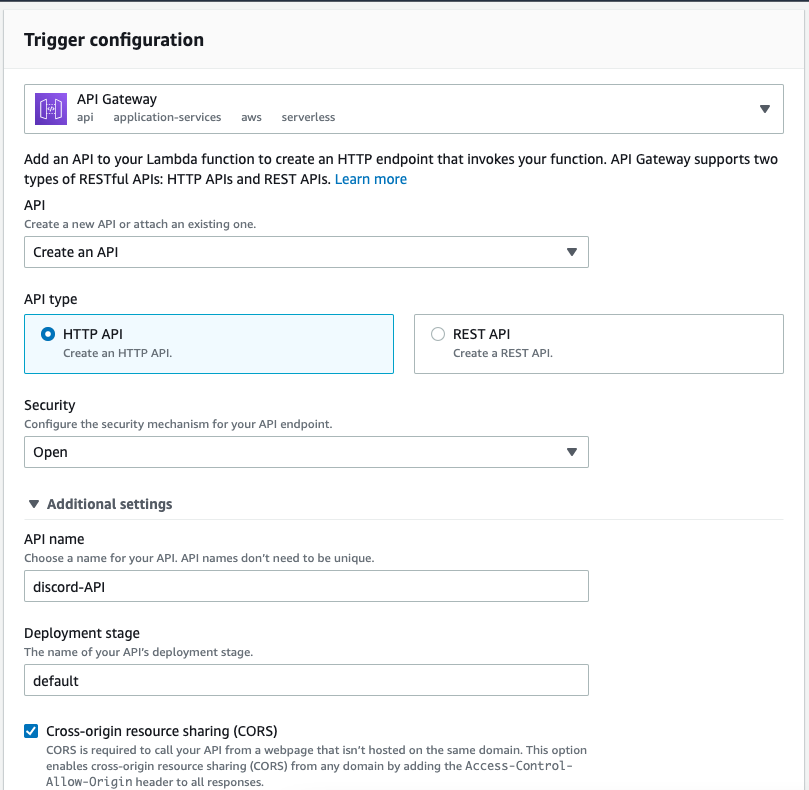


Initial Lambda dashboard

**Part 2: Add API Gateway Trigger**

**4. Add a Trigger**

* On your Lambda function's page, click **“Add trigger”**.
* Select **API Gateway**.
  + API type: **HTTP API**
  + Security: **Open** (you can restrict later)
* Enable **CORS**.

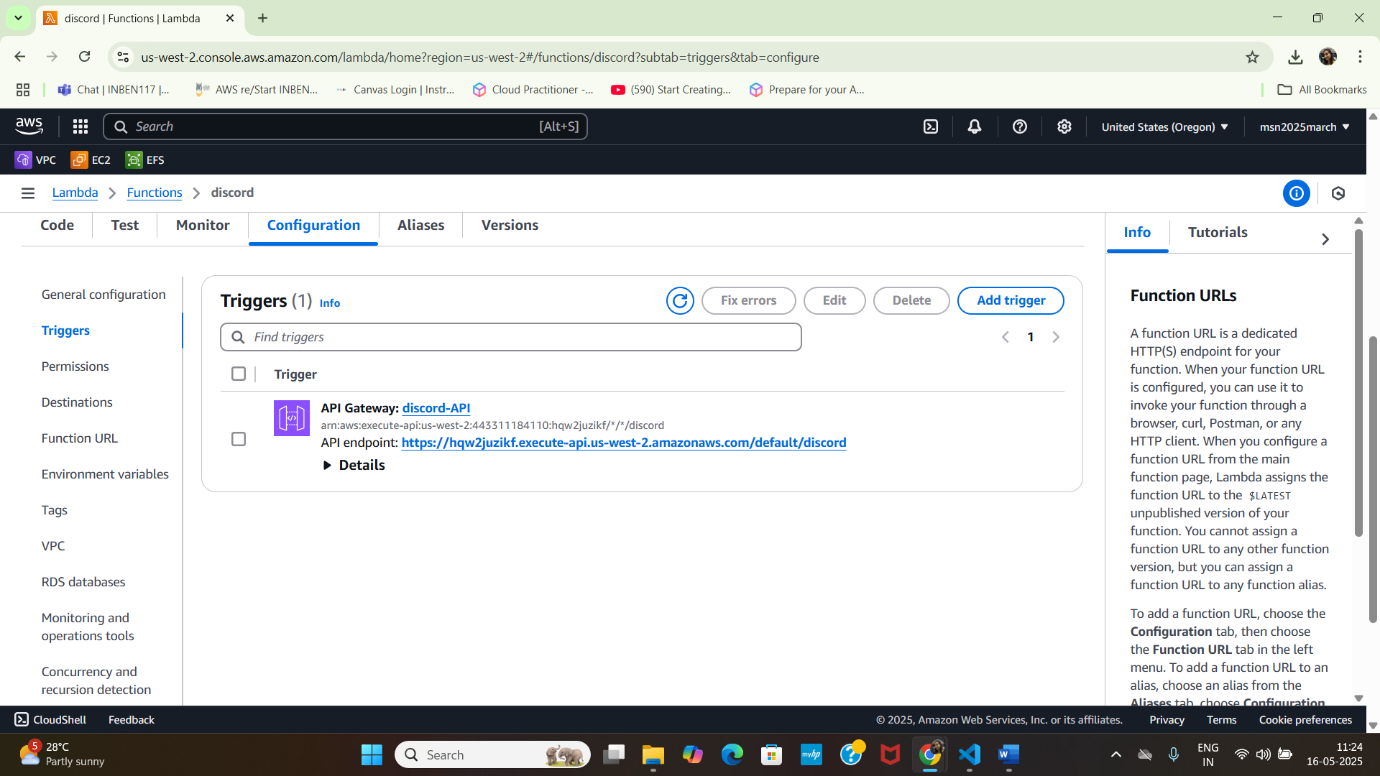


Trigger configuration

**5. Save & Get Endpoint URL**

* After the trigger is created, you’ll see an **API endpoint URL** like:
* https://xxxxxxxxxx.execute-api.region.amazonaws.com/default/discord
* Keep this URL — it will be used in the Discord Developer Portal.

The result



Trigger created

Go to the API endpoint by clicking on its URL: you should see “Hello from Lambda!”

* **Code**

It does not feel comfortable to write our code in AWS online editor. Therefore we will code locally and upload the full code to AWS Lambda.

**Part 3: Write Your Lambda Bot Code Locally**

**6. Create Project Folders**

mkdir serverless\_discord

cd serverless\_discord

mkdir lambda\_bot

cd lambda\_bot

In our local computer create project directory serverless\_discord/. Inside, create a subdirectory lambda\_bot/ and index.mjs within. Use the following code:

**7. Add Starter Lambda Code (index.mjs)**

import nacl from 'tweetnacl';

export const handler = async (event) => {

  console.log("Event:", JSON.stringify(event));

  const PUBLIC\_KEY = process.env.PUBLIC\_KEY;

  const signature = event.headers['x-signature-ed25519'];

  const timestamp = event.headers['x-signature-timestamp'];

  const body = event.body;

  if (!signature || !timestamp || !body) {

    return {

      statusCode: 400,

      body: 'Missing headers or body',

    };

  }

  const isVerified = nacl.sign.detached.verify(

    Buffer.from(timestamp + body),

    Buffer.from(signature, 'hex'),

    Buffer.from(PUBLIC\_KEY, 'hex')

  );

  if (!isVerified) {

    return {

      statusCode: 401,

      body: 'Invalid request signature',

    };

  }

  const json = JSON.parse(body);

  if (json.type === 1) {

    return {

      statusCode: 200,

      body: JSON.stringify({ type: 1 }),

    };

  }

  return {

    statusCode: 200,

    body: JSON.stringify({

      type: 4,

      data: { content: 'Hello from Lambda! (Node 22.x)' },

    }),

  };

};

**8a.** Create a package.json file with:

{

"type": "module"

}

**8b. Install Dependencies**

npm init -y

npm install tweetnacl

**Part 4: Upload to AWS Lambda**

**9. Zip Code**

**Option 1: Use PowerShell's Compress-Archive Command**

Run this command inside your lambda\_bot folder:

Compress-Archive -Path \* -DestinationPath ../lambda\_bot.zip

This will create lambda\_bot.zip one level up, containing everything in your current directory, including node\_modules.

**AWS**

**10. Upload via AWS Console**

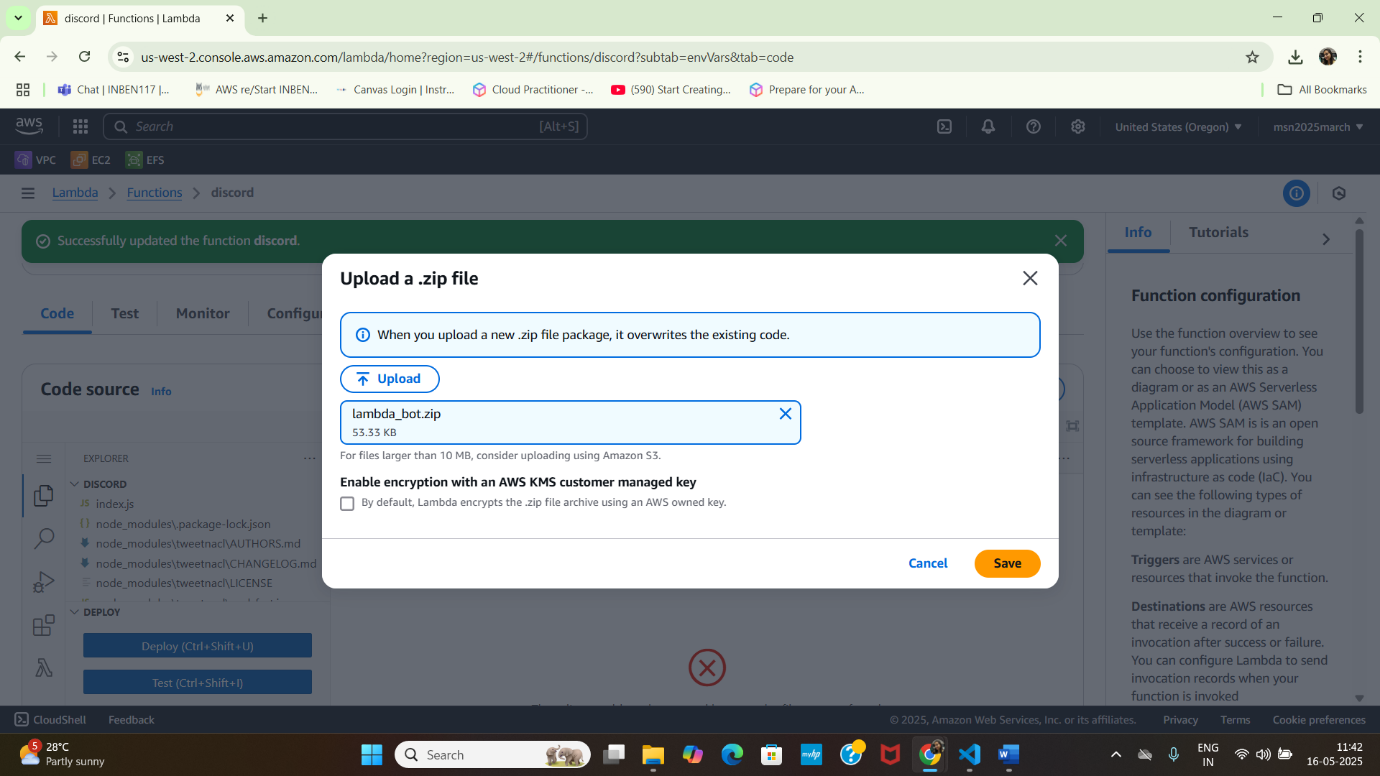
* Go to Lambda > Code tab > **Upload from .zip file**
* Upload lambda\_bot.zip and click **Deploy**

**OR**

**10B. Upload via AWS CLI**

aws lambda update-function-code \  
 --function-name discord \  
 --zip-file fileb://../lambda\_bot.zip

Upload the zip in Lambda dashboard — section “Code”.



Lambda dashboard.

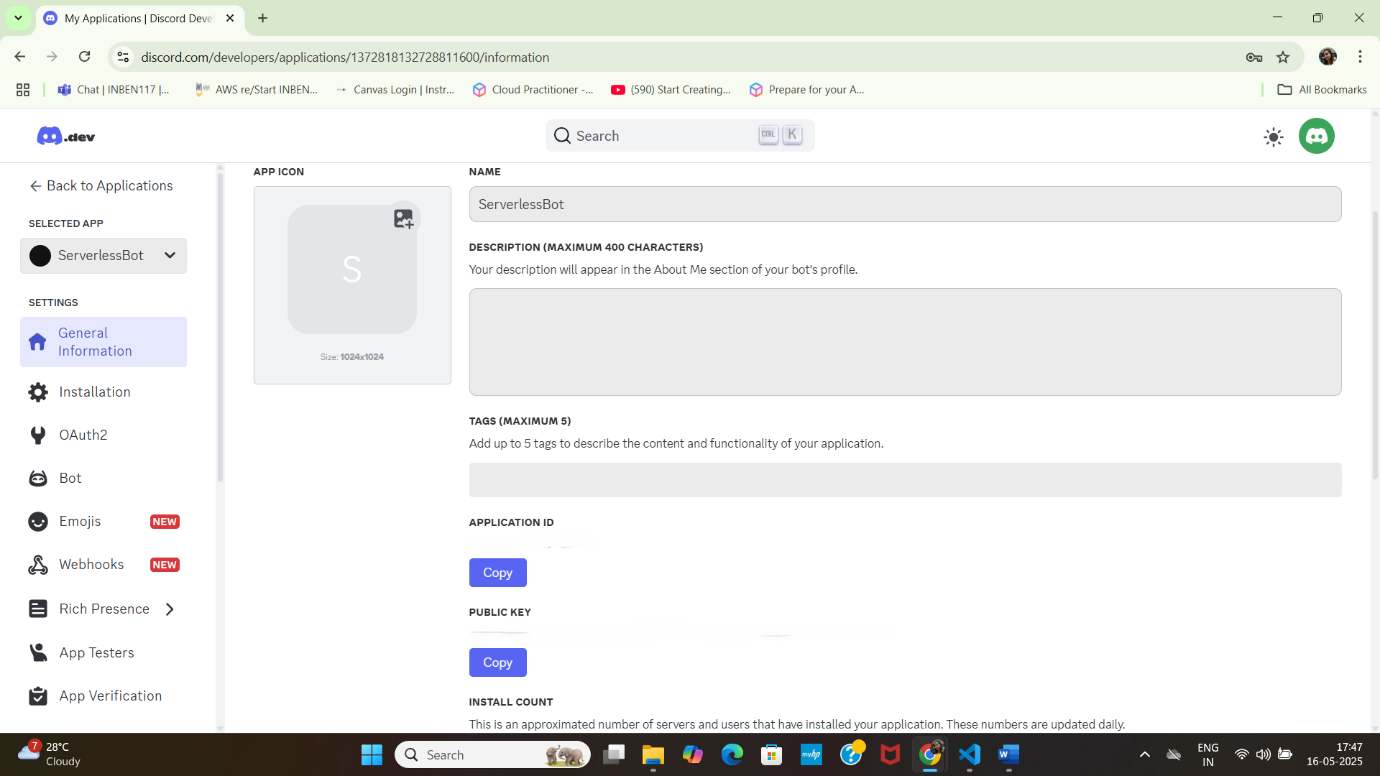
* **Discord Developer Portal**

**11. Create a Discord App**

* Visit: https://discord.com/developers/applications
* Click **“New Application”**
* Name it something like ServerlessBot, then click **Create**

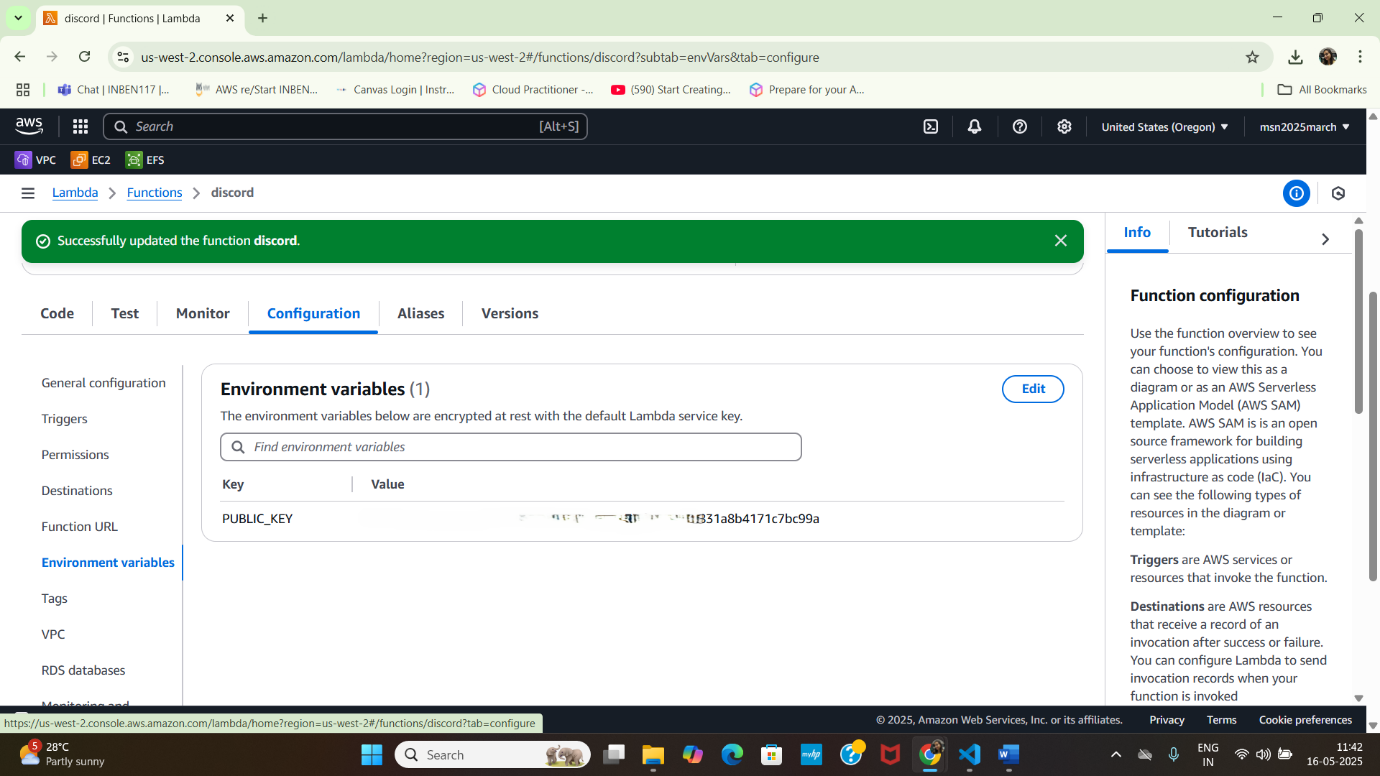
**12. Get Public Key**

* In the app dashboard, click **“General Information”**
* Copy the **Public Key** (you'll need it for Lambda)



**13. Set Environment Variable in Lambda**

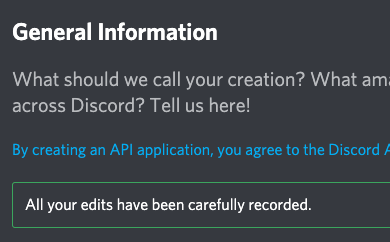
* AWS Console → Lambda → Configuration → Environment Variables
* Add:
  + **Key:** PUBLIC\_KEY
  + **Value:** your Discord Public Key (no quotes)



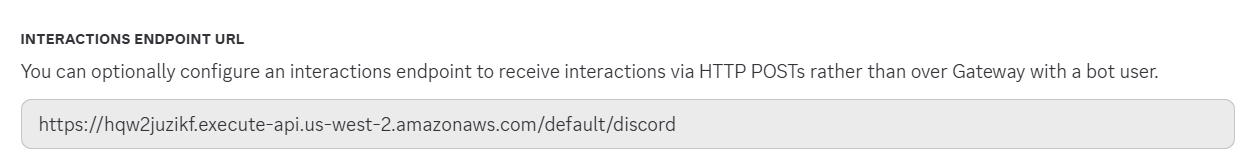
**14. Add API Endpoint to Discord**

* **Go to** your Discord app in the Developer Portal
* Navigate to the **General Information** tab.
* Go to **"Interactions Endpoint URL"** in the Discord Developer Portal
* Paste your API Gateway URL
* Click **Save**

**If everything is correct, you’ll see a green success message.**



All your edits have been carefully recorded.



My real AWS API Gateway URL.

and click Save!

Saving performs Discord-side check of their requirements — they send a set of POST requests to your endpoint.

**Congrats! You’ve connected Discord to AWS Lambda.**

You now have:

* A working serverless function
* A public endpoint
* Discord’s Interactions verifying and invoking your code.

**Part 5:** **Register a Slash Command (/foo → "bar")**

**Step 1: Create Command Registration Script Locally**

Navigate to your project root (e.g. serverless\_discord/) and create a new folder for the registration script:

mkdir register\_commands

cd register\_commands

**Step 2: Create register.mjs**

Create a file called register.mjs inside register\_commands/

**register.mjs**

import axios from 'axios';

import dotenv from 'dotenv';

import { commands } from './commands.js';

dotenv.config();

const DISCORD\_TOKEN = process.env.DISCORD\_TOKEN;

const CLIENT\_ID = process.env.CLIENT\_ID;

const GUILD\_ID = process.env.GUILD\_ID;

console.log('Registering commands:', JSON.stringify(commands, null, 2));

axios

  .put(

    `https://discord.com/api/v10/applications/${CLIENT\_ID}/guilds/${GUILD\_ID}/commands`,

    commands,

    {

      headers: {

        Authorization: `Bot ${DISCORD\_TOKEN}`,

        'Content-Type': 'application/json',

      },

    }

  )

  .then((response) => {

    console.log('✅ Commands registered successfully:', response.data);

  })

  .catch((error) => {

    console.error('❌ Error registering slash commands:', error.response?.data || error.message);

  });

**Step 3: Create .env File**

Still inside register\_commands/, create a .env file and add your actual values:

**.env file**

DISCORD\_TOKEN= an38VjI6koHx5pxWzDzNiBgeSMHgFf715cfYw8

CLIENT\_ID=1372832728811600

GUILD\_ID=137221798237224

DISCORD\_TOKEN =your\_discord\_bot\_token\_here

CLIENT\_ID=your\_discord\_app\_id\_here

GUILD\_ID=your\_guild\_id\_here

To get those:

* **DISCORD\_TOKEN** → Discord Developer Portal → Bot → *Reset Token* if needed
* **CLIENT\_ID** → Developer Portal → General Information→ **Application ID**
* **GUILD\_ID** → Right-click your server in Discord → *Copy ID* (Developer Mode must be ON)

**Step 4: Install Dependencies**

From inside register\_commands/, run:

npm init -y

npm install axios dotenv

**Step 5:Update package.json:**

Make sure your package.json file in the register\_commands folder includes the following line:

{

"type": "module"

}

**Step 6:Check Bot Permissions**

Ensure that the bot has been properly invited to the guild (server) with the required permissions.

To invite the bot with the correct permissions, make sure you're using an invite URL with the applications.commands and bot scopes, along with the necessary permissions to register commands.

**1.Copy this URL** into your browser:

<https://discord.com/oauth2/authorize?client_id=YOUR_APP_ID&scope=bot%20applications.commands&permissions=0>

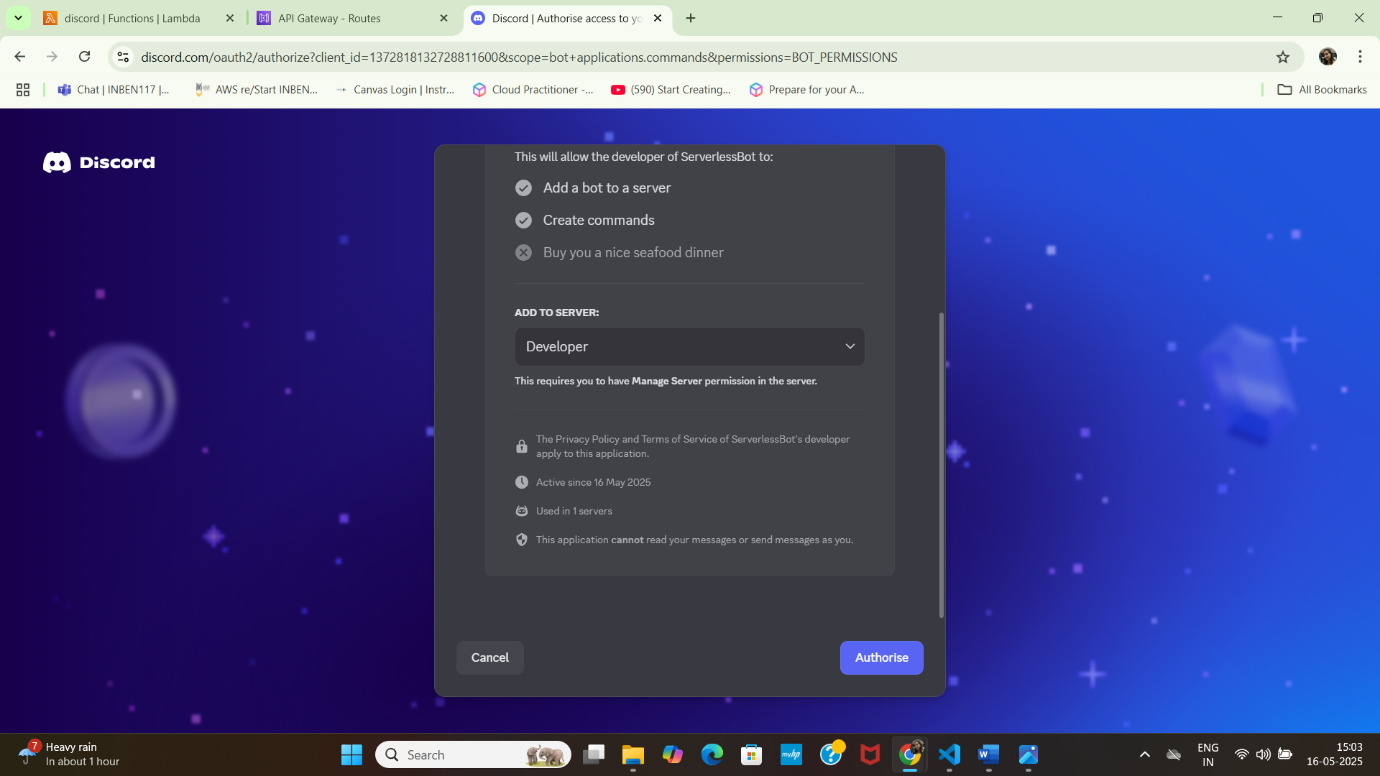
The permissons=PERMISSION\_INTEGER should be replaced with the correct integer value for the bot's required permissions. At a minimum, the bot needs applications.commands scope, which allows it to register slash commands.

**2.Paste the URL** into your browser and hit **Enter**

**3.Select the correct Discord server (guild)** where you want the bot to be added

**4**.Click **Authorize**

You should now see the bot appear in the server — or already be there if previously added



After you've authorized the bot, you can run your register.mjs script again to register commands.

**Step 7: Create commands.js file**

// register\_commands/commands.js

import { SlashCommandBuilder } from '@discordjs/builders';

const commands = [

  new SlashCommandBuilder()

    .setName('foo')

    .setDescription('Replies with bar')

    .toJSON()

];

export { commands };

Note the use of SlashCommandBuilder() — it builds a properly formatted command including all required fields like type.

**Ensure the Package is Installed:**

Run the following command from your project root to install the dependencies listed in package.json:

npm install

npm list @discordjs/builders

If it's installed correctly, you should see output similar to this:

serverless\_discord@1.0.0 E:\AWS Cloud Practitioner\Exam Prep\serverless\_discord

└── @discordjs/builders@0.13.0

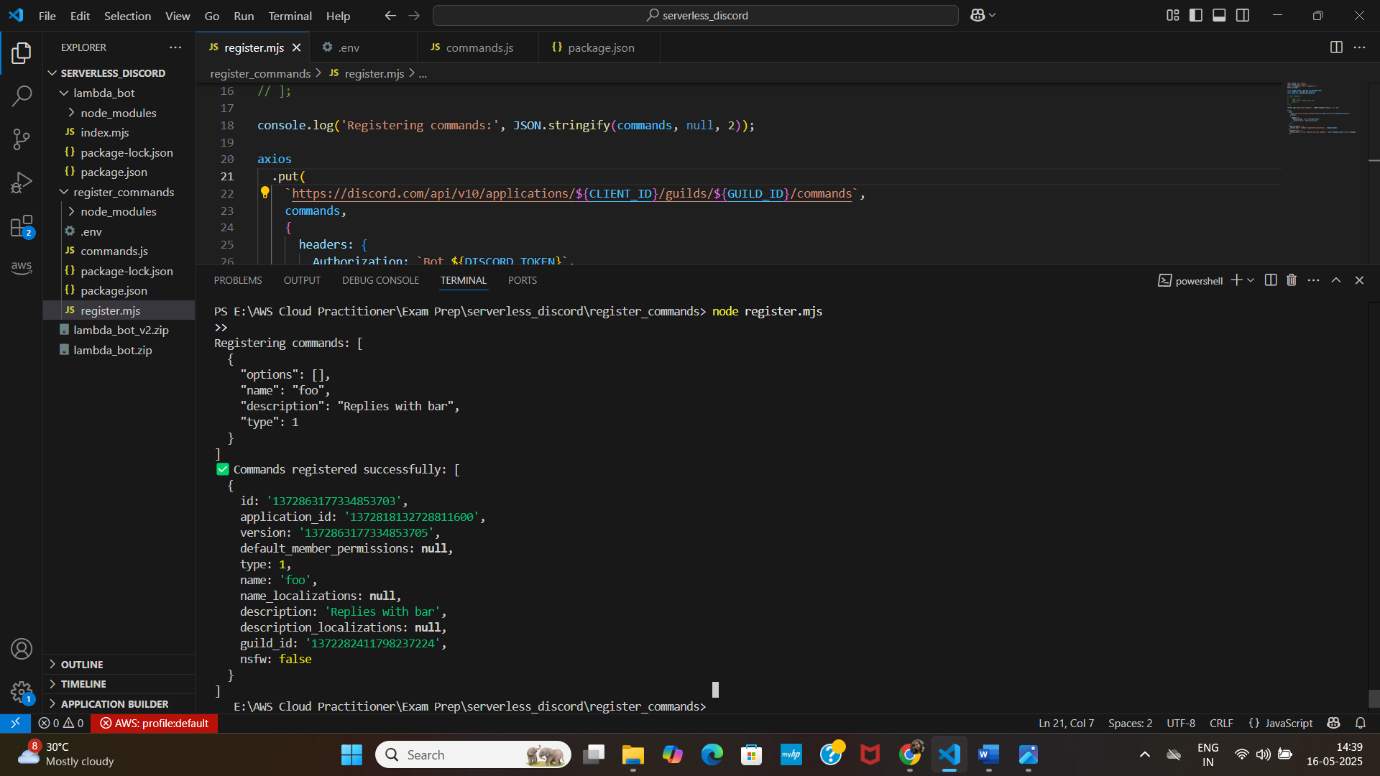
**Step 8: Run the Registration Script**

node register.mjs

**Expected Output:**

Registering commands: [ ... ]

Slash commands registered successfully.



**Part 6: Handle Incoming Slash Commands**

**1. Set up your bot to listen and respond**

Create a file like bot.mjs (or use your existing bot entry point) with the following:

import { Client, GatewayIntentBits, Events } from 'discord.js';

import dotenv from 'dotenv';

dotenv.config();

const client = new Client({ intents: [GatewayIntentBits.Guilds] });

client.once(Events.ClientReady, () => {

  console.log(`🤖 Logged in as ${client.user.tag}`);

});

client.on(Events.InteractionCreate, async interaction => {

  if (!interaction.isChatInputCommand()) return;

  if (interaction.commandName === 'foo') {

    // Respond quickly to avoid timeout

    await interaction.reply('Hello from bot!');

  }

});

client.login(process.env.DISCORD\_TOKEN);

**2. Ensure discord.js is installed**

If not yet installed:

npm install discord.js

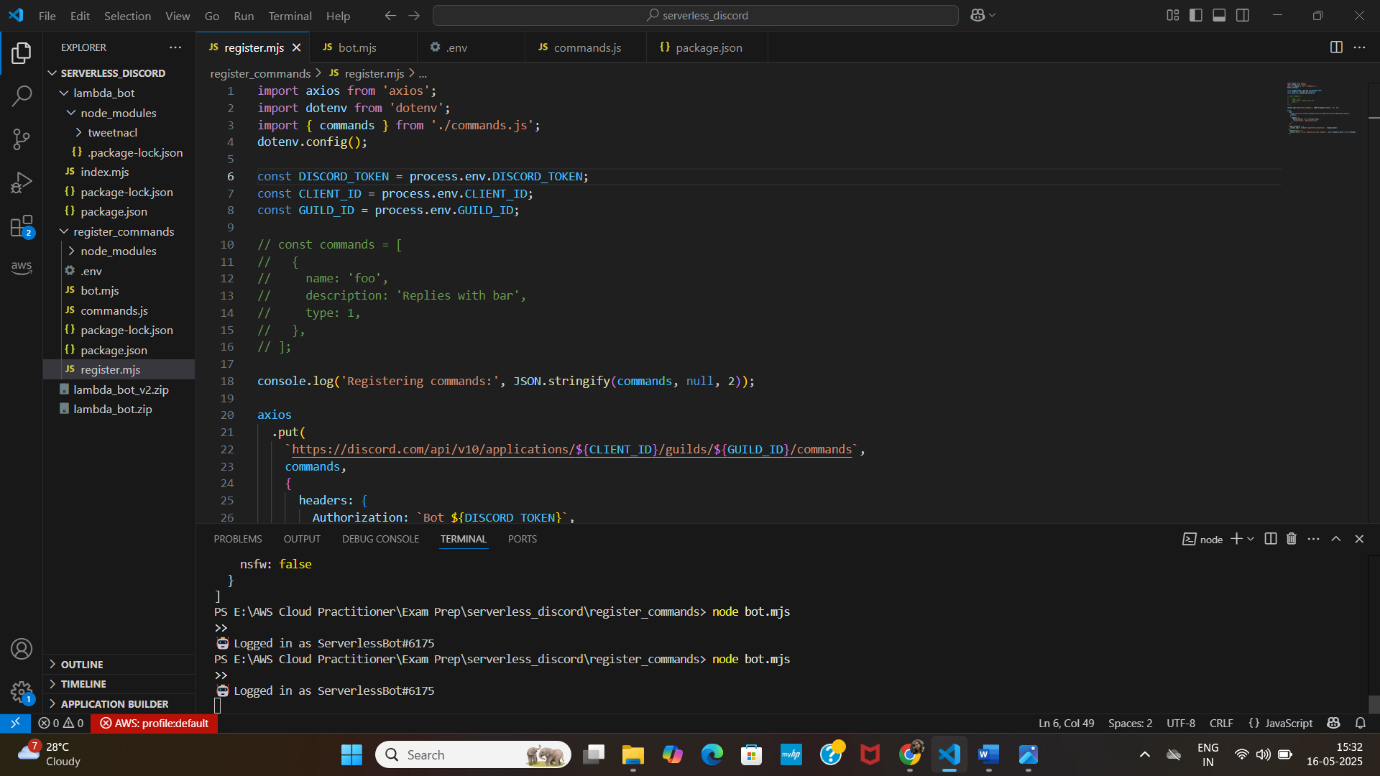
**3. Run your bot**

From the same folder:

node bot.mjs

You should see something like:

🤖 Logged in as YourBotName#1234



**4. Test in Discord**

* Go to the server where you added the bot.
* In any text channel, type /foo — and your bot should reply with bar.

**Optional: Add More Commands**

You can add more commands in commands.js using:

new SlashCommandBuilder()

.setName('hello')

.setDescription('Replies with world')

And then handle them in bot.mjs:

if (interaction.commandName === 'hello') {

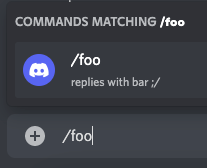
await interaction.reply('world');

}

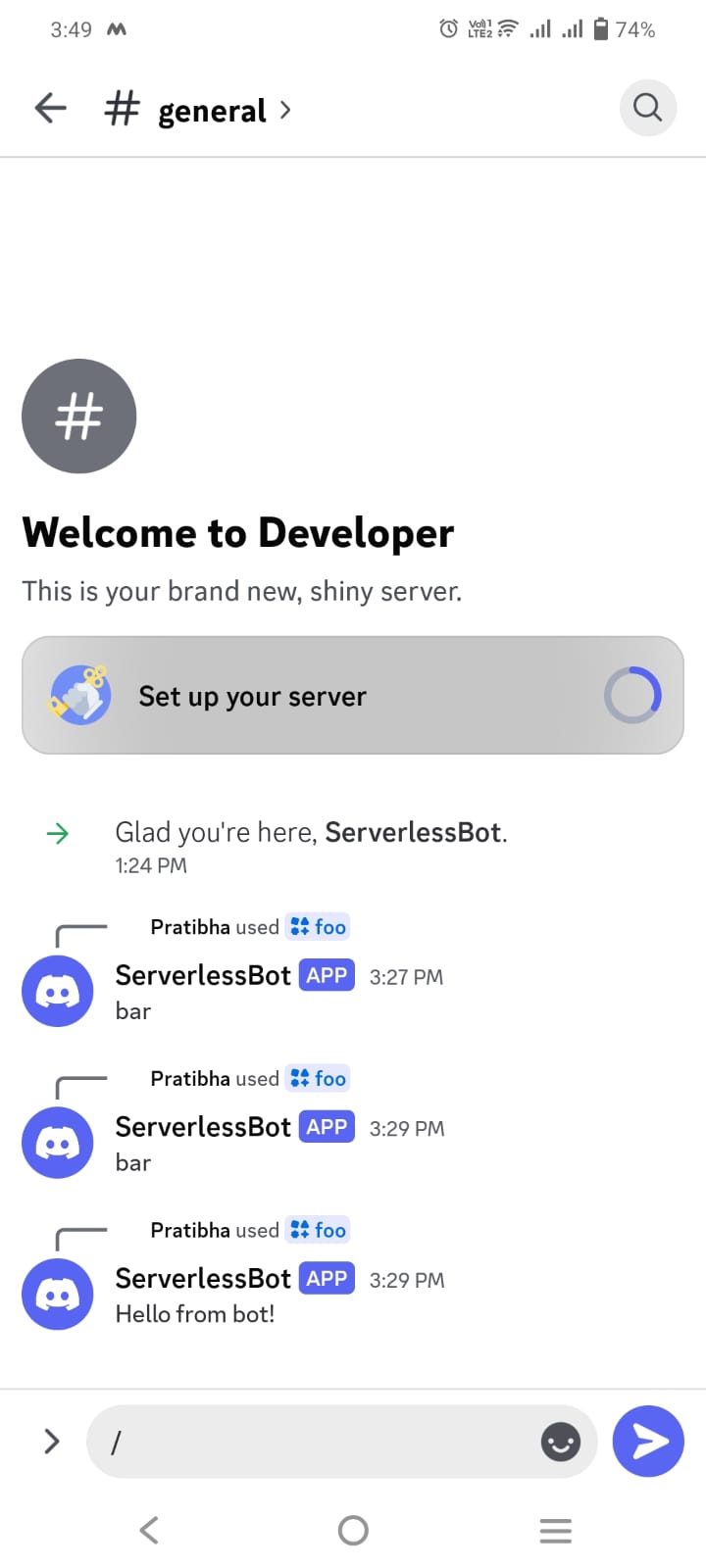
**Final Test**

1. Clear the Interactions Endpoint URL (if set)
2. Ensure the bot is running (node bot.mjs)
3. In Discord, type /foo
4. The bot should respond with bar within 1–2 seconds

No error means success! Go quickly to your Discord Guild chat, start typing /foo and press Enter.



Drumroll…



BAR! And Hello from bot!

Our serverless bot is now up and running waiting for events.

**Conclusion**

Through this project, we transitioned away from traditional always-on servers to a fully event-driven model. AWS Lambda executes lightweight bot logic only when needed, while API Gateway provides a secure and scalable entry point for incoming Discord interaction requests. Payload validation using tweetnacl, secure secret management via environment variables, and structured slash command registration ensure both security and maintainability.

**Key Takeaways**

* **Serverless architecture is ideal for stateless, event-driven bots** that don’t require persistent connections.
* **Cost remains very low**—even for moderately active bots—thanks to AWS Free Tier and Lambda’s pay-per-use model.
* **Discord’s Interactions Endpoint** model pushes developers to create more secure and scalable bots.
* You gain **hands-on experience with AWS Lambda, API Gateway, Secrets management, and cloud debugging tools** like CloudWatch.

**Limitations**

* Not suitable for bots needing **real-time message listening** or **voice interaction** (which require persistent connections).
* Requires more **deployment discipline** (e.g., zipping/uploading code or using CI/CD) compared to traditional server models.
* Relies on **vendor-specific services**, introducing some level of **vendor lock-in**.

**Future Enhancements**

* Add a **CI/CD pipeline** using GitHub Actions or AWS CodePipeline.
* Use **Step Functions** for multi-step bot logic workflows.
* Extend the bot to handle more slash commands or external APIs (e.g., weather, AI replies).
* Integrate **SSM Parameter Store** to optimize cost even further.

**Additional Sources:**

* Code with a full solution is accessible at my github repo: <https://github.com/Pmnikam/discord-aws-lambda-project.git>
* Another well made a tutorial on the subject, but with few differences (i.e. uses Python; is more elaborate): <https://oozio.medium.com/serverless-discord-bot-55f95f26f743>
* Discord Developer Docs <https://discord.com/developers/docs/interactions/application-commands> (check also “Receiving and Responding” section)