



# Serverless App using REST API



## Serverless App with REST API

👤 Created by	👤 Niyush Bjr
⌚ Created time	@26 July 2025 00:21
⭐ level	Intermediate
✍ Description	Building a Serverless Application with REST API
☁ AWS Services	API Gateway    DynamoDB    Lambda    S3    SQS
☰ Other tools	VsCode
🔧 Category	Compute
☕ Buy-Me-a-Coffee	<a href="http://buymeacoffee.com/niyushbjr1L">http://buymeacoffee.com/niyushbjr1L</a>

Overall Architecture.

Step 1 : Create the SQS queue.

Step 2 : Create the DynamoDB table.

Step 3 : Create the Lambda Functions.

    Creating the first lambda function.

    Creating the Second Lambda Function.

Step 4 : Deploy and test the application.

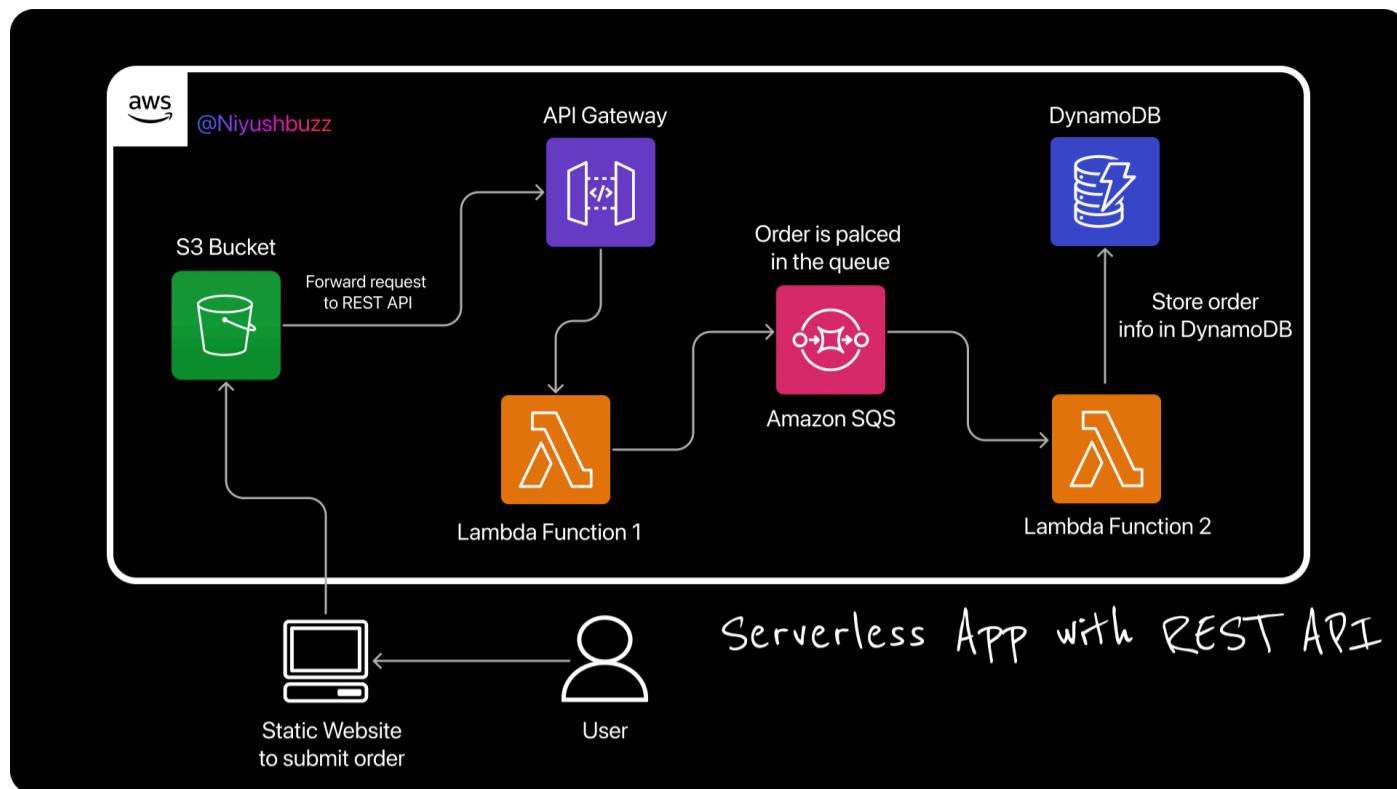
Step 5 : Creating the REST API

Step 6 : Create the S3 bucket

Step 7 : Testing time

Step 8 : Delete the resources

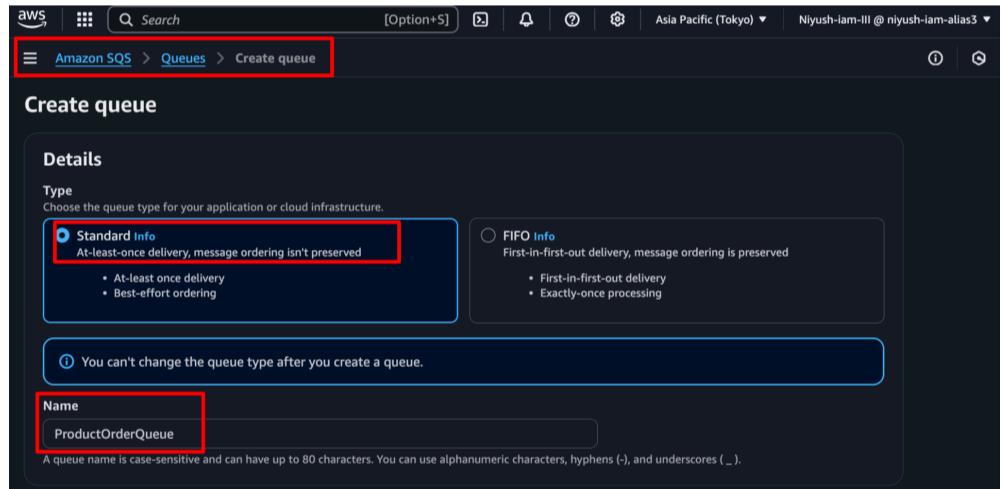
## Overall Architecture.



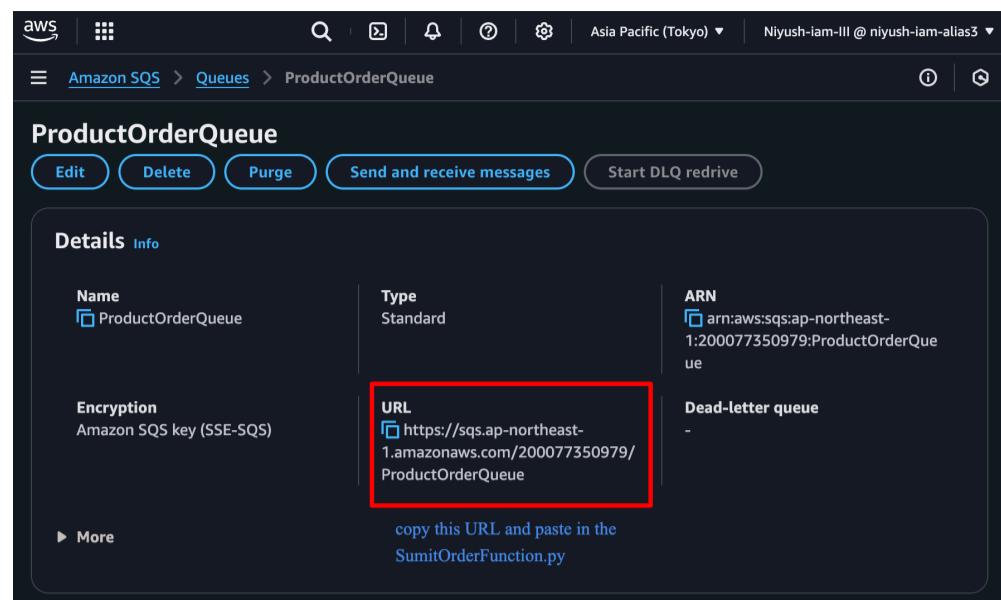
## Step 1 : Create the SQS queue.

- Head to SQS console.
- Queue Type : Standard queue.
- Name : `ProductOrderQueue`
- Leave all the Remaining setting to `Default` and then click on `create queue`
- Copy the URL of the queue.  
(this will be used later on with lambda function)

▼ [Image] Create the queue



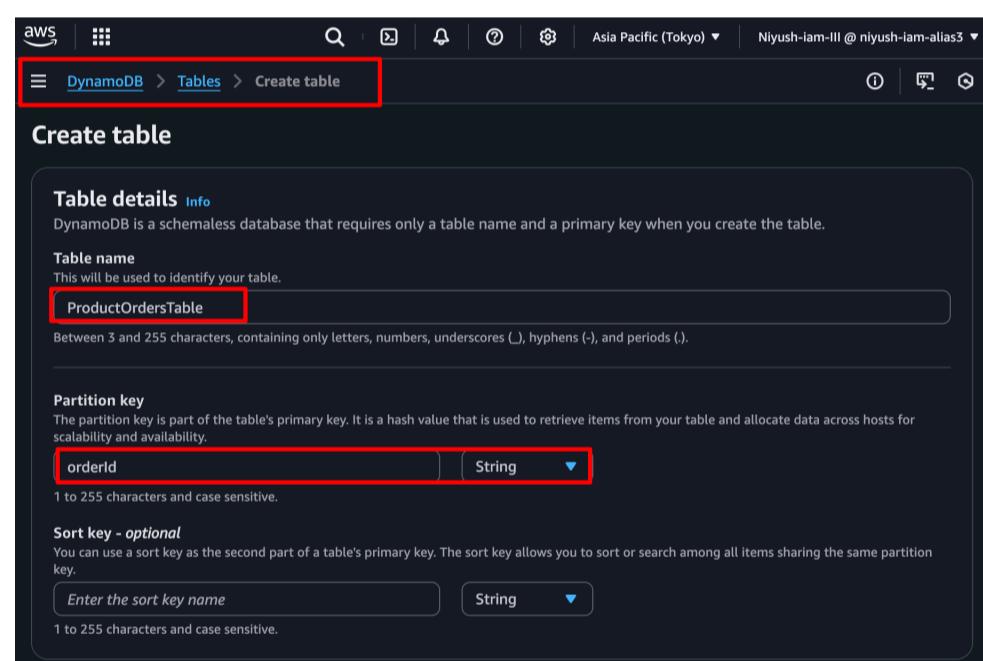
Queues (1)	
<a href="#">Edit</a> <a href="#">Delete</a> <a href="#">Send and receive messages</a> <a href="#">Actions ▾</a> <a href="#">Create queue</a>	
<input type="text"/> Search queues by prefix	
<b>Name</b> ▲ <b>Type</b> ▼ <b>Created</b> ▼ <b>Messages available</b>	
<input type="radio"/> <a href="#">ProductOrderQueue</a>	Standard 2025-08-03T19:53+09:00 0



## Step 2 : Create the DynamoDB table.

- Head to DynamoDB console → [Create Table](#)
- Table Name : [ProductOrdersTable](#)
- Primary Key : orderId
- Leave the rest to default and hit [Create Table](#)

▼ [Image] Create a dynamoDB table



## Step 3 : Create the Lambda Functions.

### Creating the first lambda function.

- Head over to Lambda Console → [Create Function](#)
- Function Name : [SubmitOrderFunction](#)
- Runtime : Python 3.9

▼ [Image] Creating the lambda Function Image

### ▼ Python Code Here ↓

```
import json
import boto3

sns = boto3.client('sns')
queue_url = 'YOUR_SQS_QUEUE_URL'

def lambda_handler(event, context):
```

```

try:
    order_details = json.loads(event['body'])
    response = sqs.send_message(QueueUrl=queue_url, MessageBody=json.dumps(order_details))
    return {'statusCode': 200, 'headers': {'Access-Control-Allow-Origin': '*', 'Access-Control-Allow-Headers': 'Content-Type', 'Access-Control-Allow-Methods': 'OPTIONS,POST'}, 'body': json.dumps({'message': 'Order submitted to queue successfully'})}
except Exception as e:
    return {'statusCode': 400, 'body': json.dumps({'error': str(e)})}

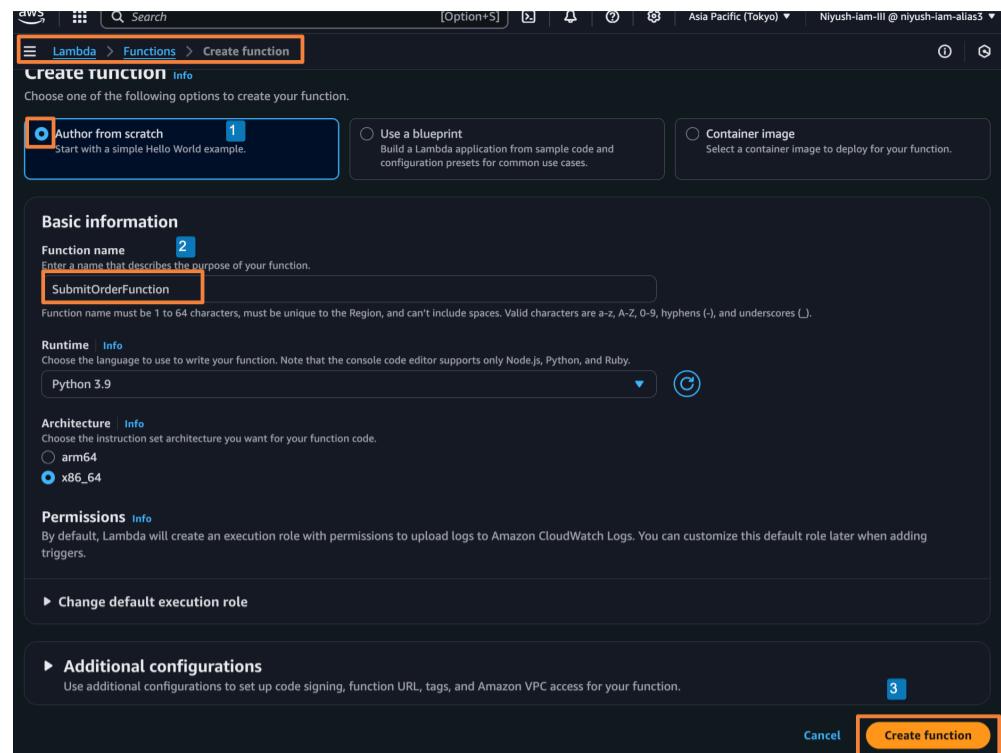
```

- Add the URL of the SQS queue to the python code.
- Add permission to the Lambda function
  - `AmazonSQSFullAccess`
- Then finally hit Deploy
- Once the Function has been successfully Created Check or test the function.

#### ▼ Test Code for Lambda Function 1.

```
{
  "body": "{\"productName\":\"THIS_IS_A_SAMPLE_TEST\",\"QUANTITY\":100}"
}
```

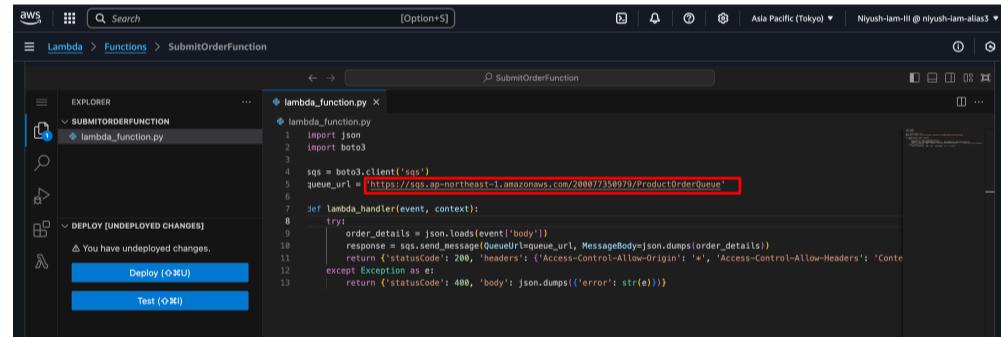
- Go the SQS and poll for messages
  - there you should see a message waiting the queue.



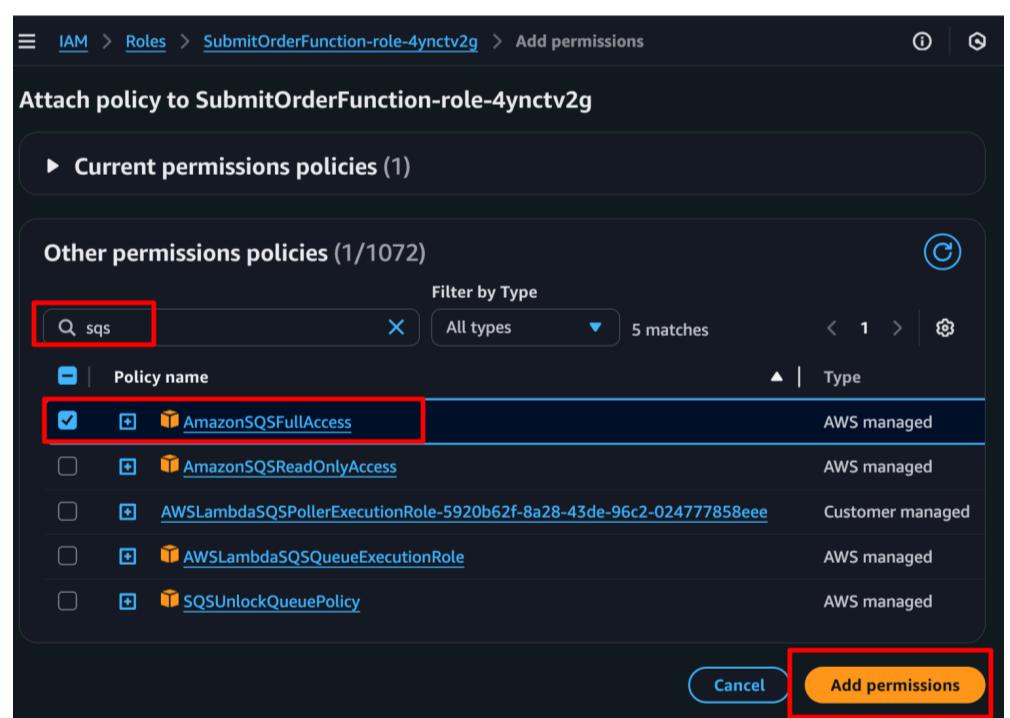
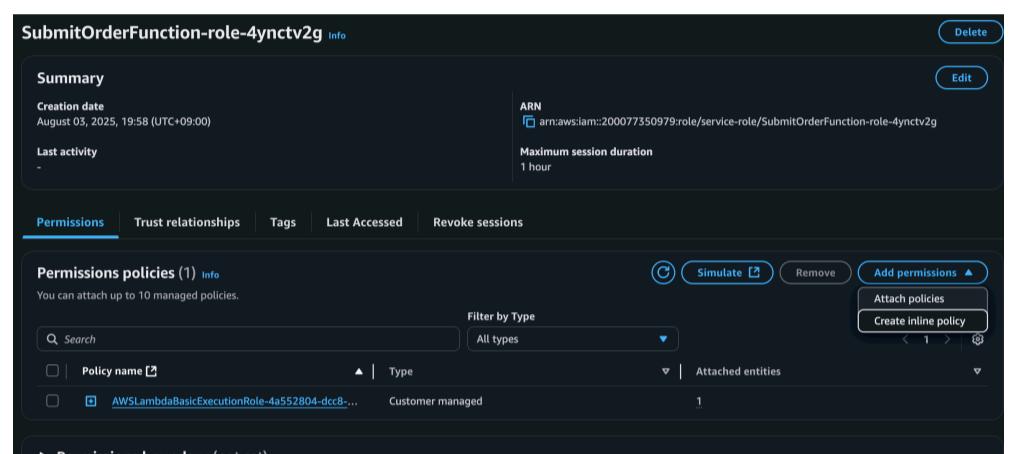
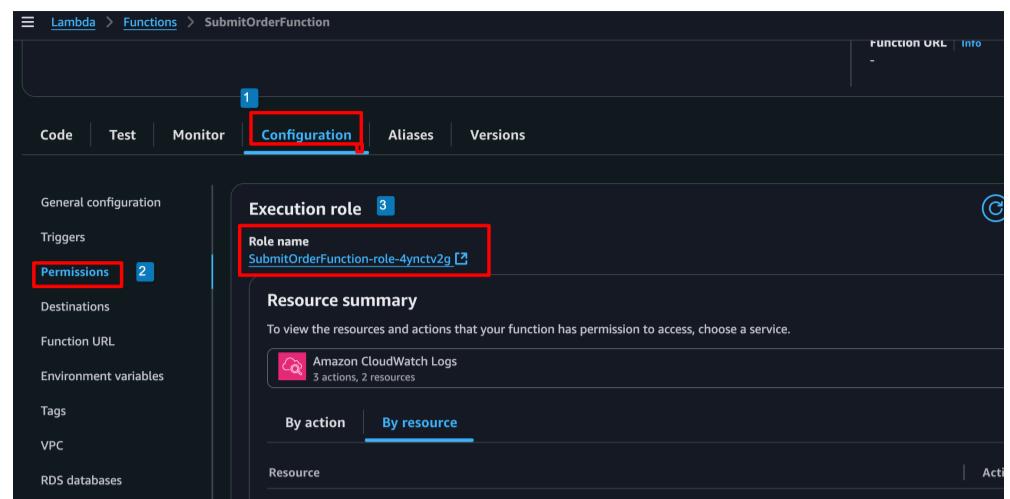
Creating the lambda function

jj

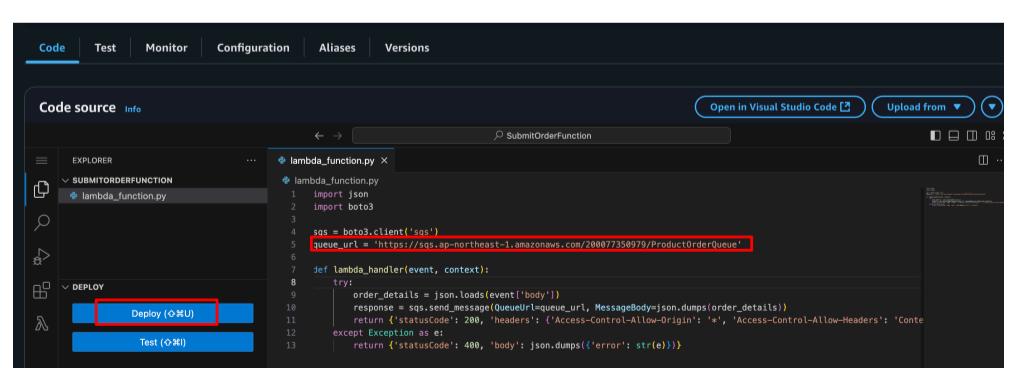
▼ [Image] adding code to the lambda function and updating the URL

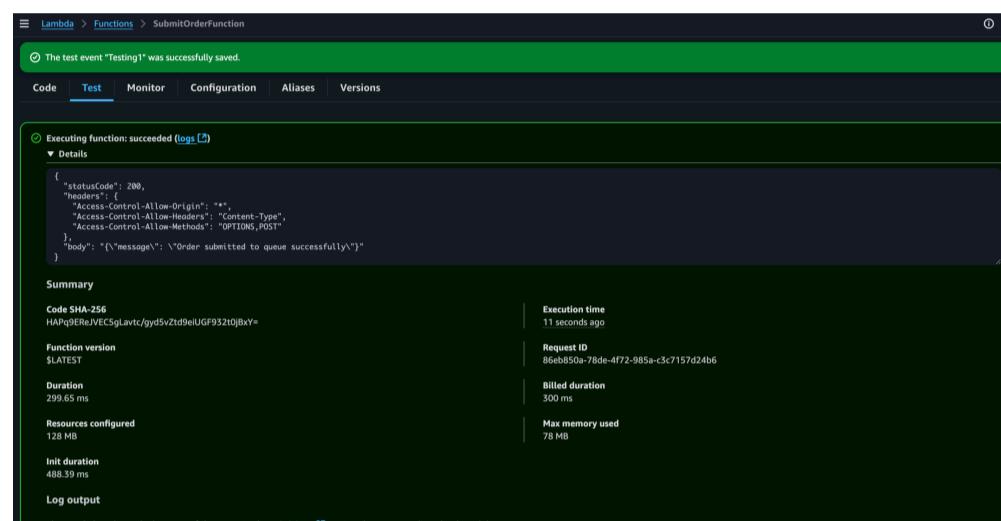
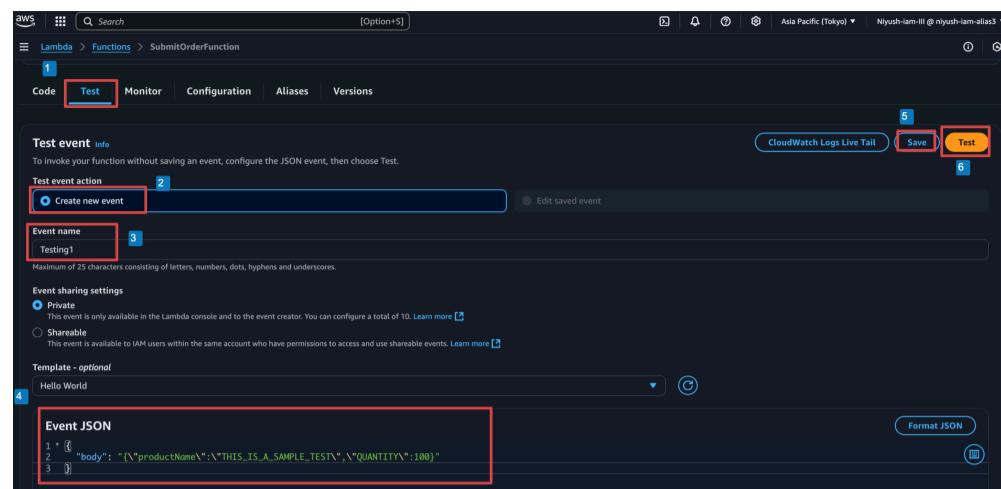


▼ [Image] adding permission to Lambda function.

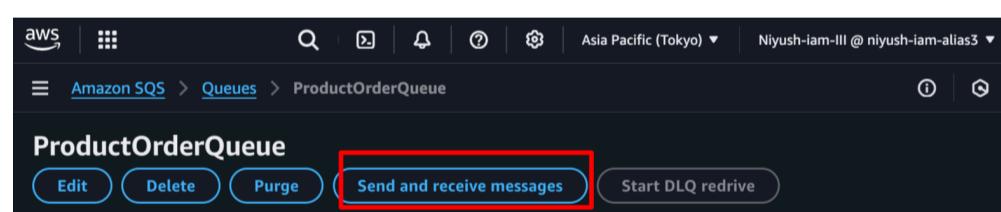


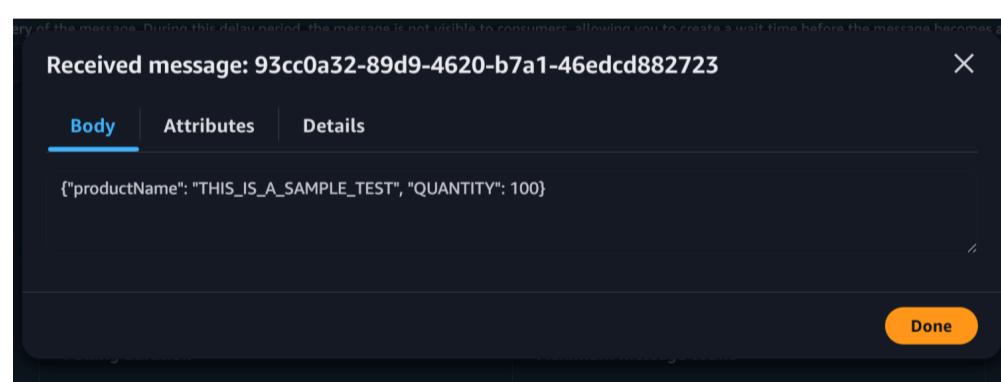
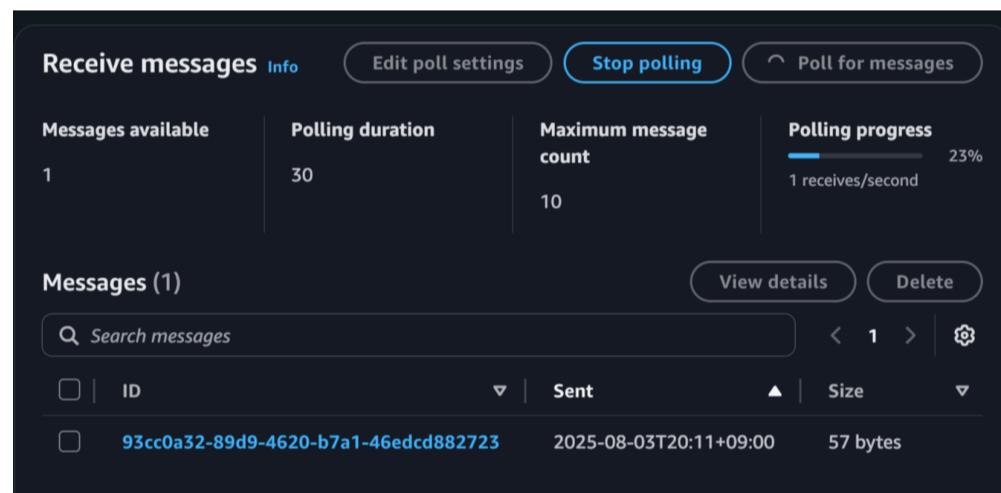
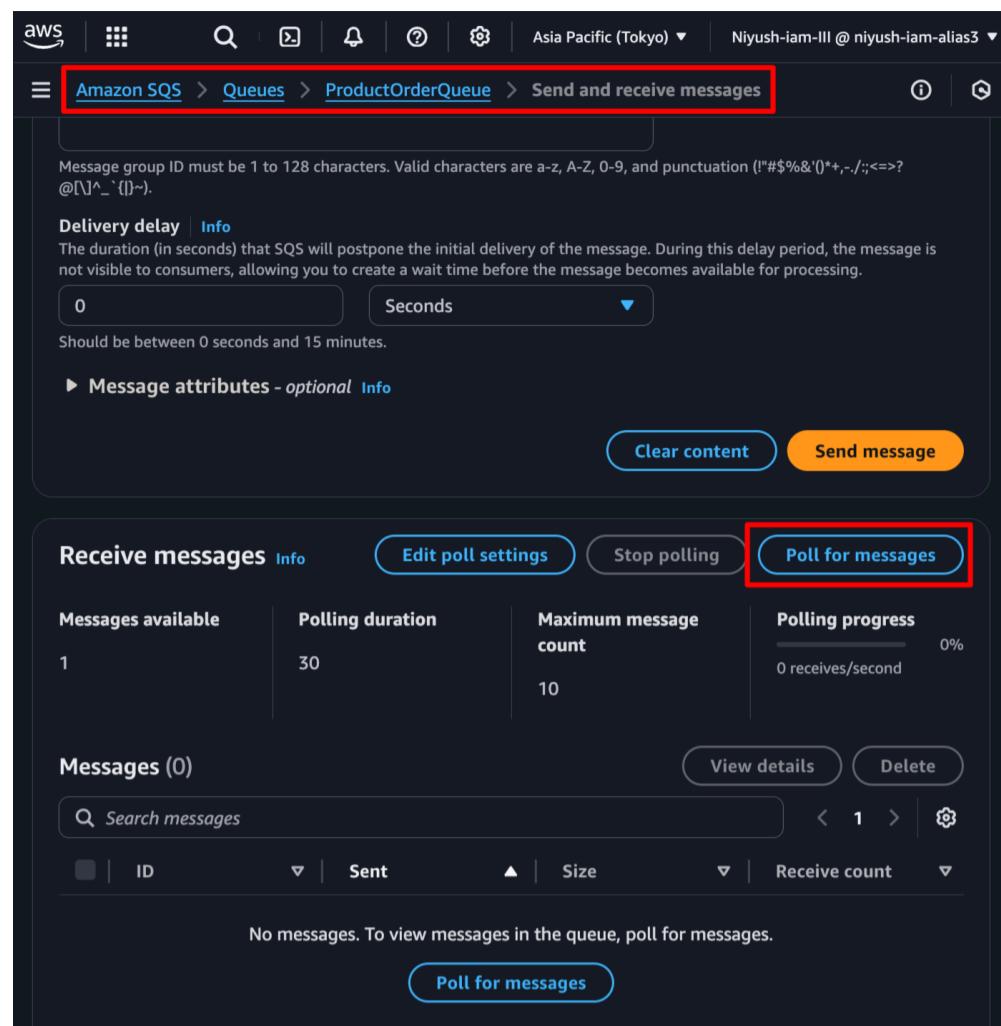
▼ [Image] Deploy and test the lambda function





### ▼ [Image] Poll for message in the SQS queue





## Creating the Second Lambda Function.

- Go the Lambda console and create another function.
- Name : `ProcessOrderFunction`  
Runtime : Python 3.9

▼ [Image] Making the second lambda function

## ▼ Code for ProcessOrderFunction.

```

import json
import boto3
from datetime import datetime

dynamodb = boto3.resource('dynamodb')
table_name = 'YOUR_DYNAMODB_TABLE_NAME'
table = dynamodb.Table(table_name)

def lambda_handler(event, context):
    for record in event['Records']:
        message_body = json.loads(record['body'])

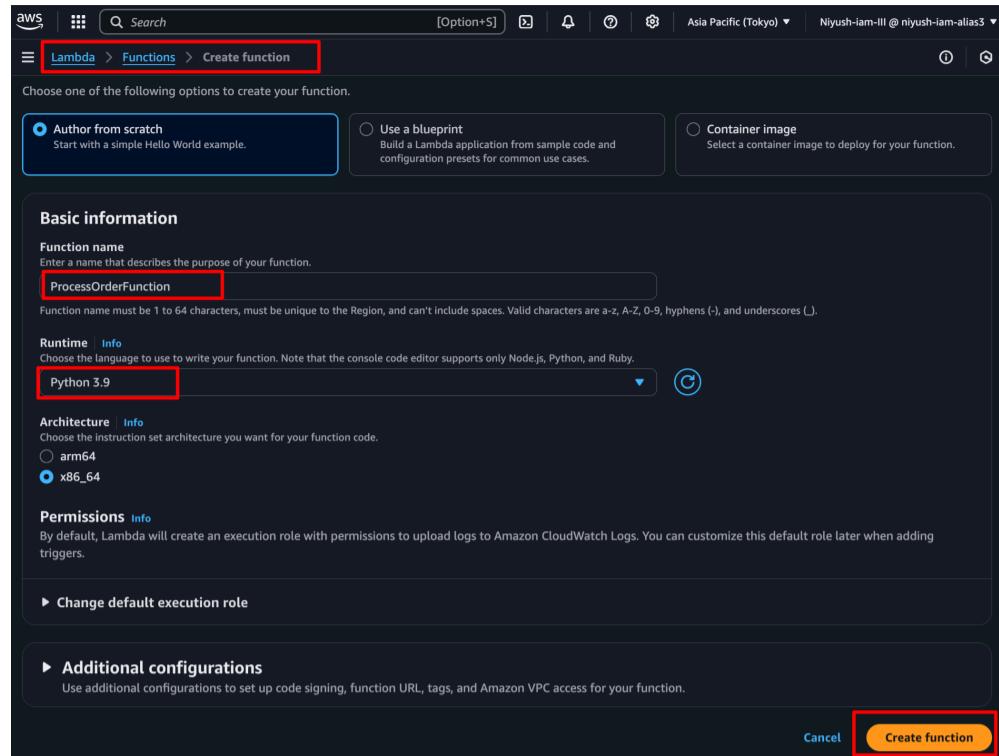
        item = {'orderId': record['messageId'], 'productName': message_body['productName'], 'quantity': message_body['quantity'], 'orderDate': datetime.now().isoformat()}

        table.put_item(Item=item)

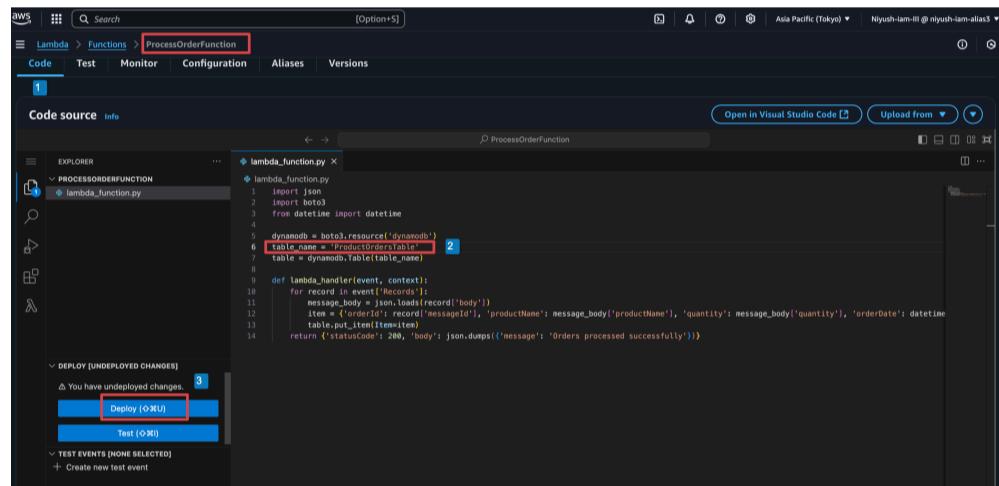
    return {'statusCode': 200, 'body': json.dumps({'message': 'Orders processed successfully'})}

```

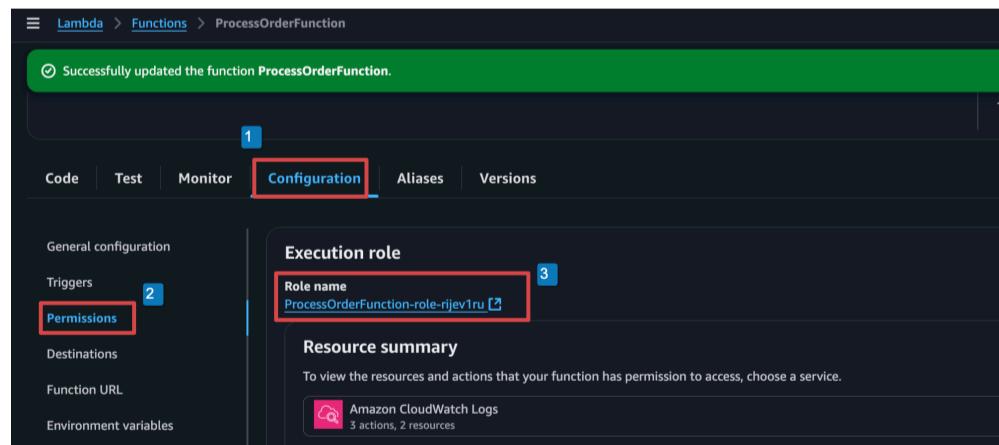
- In the above make sure to replace the table\_name with the actual DynamoDB table name.
- Add the following permission policies to the execution role.
  - AmazonSQSFullAccess
  - AmazonDynamoDBFullAccess

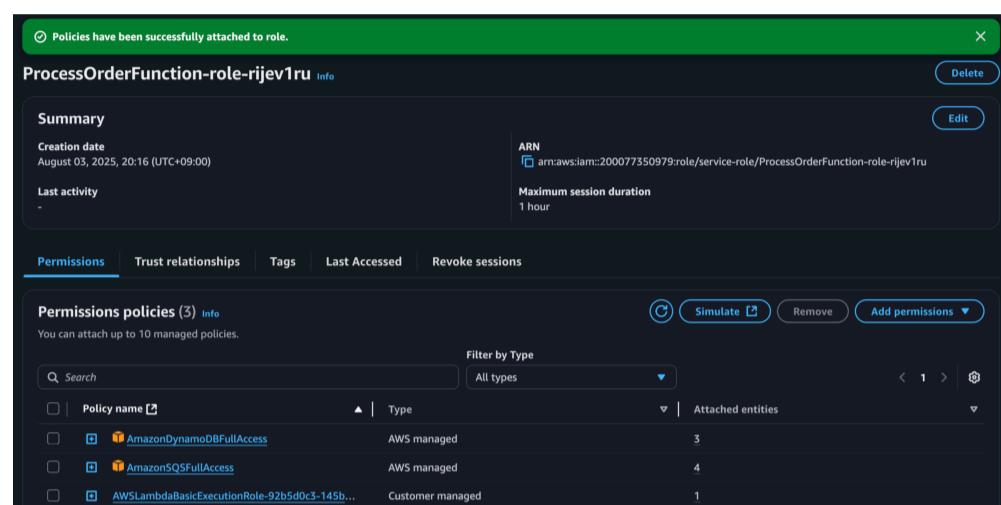
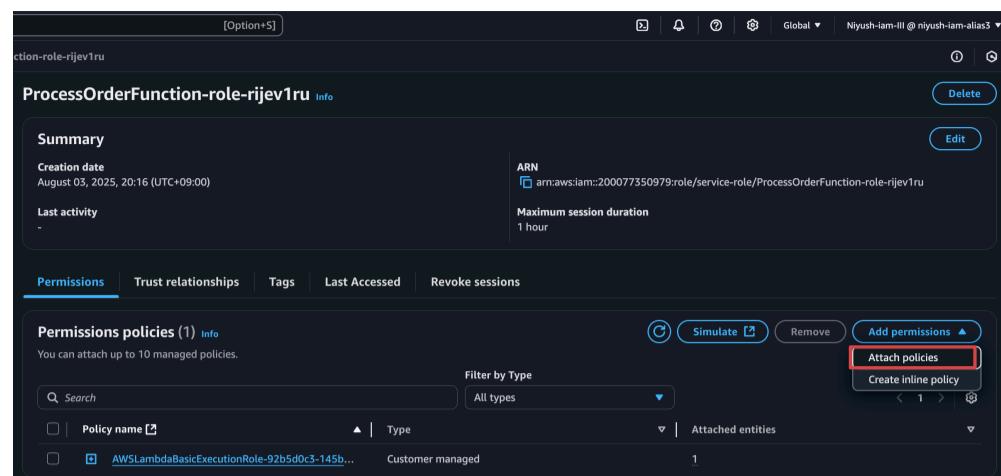


## ▼ [Image] Replacing the table name with actual table name



## ▼ [Image] Adding permission to the function2

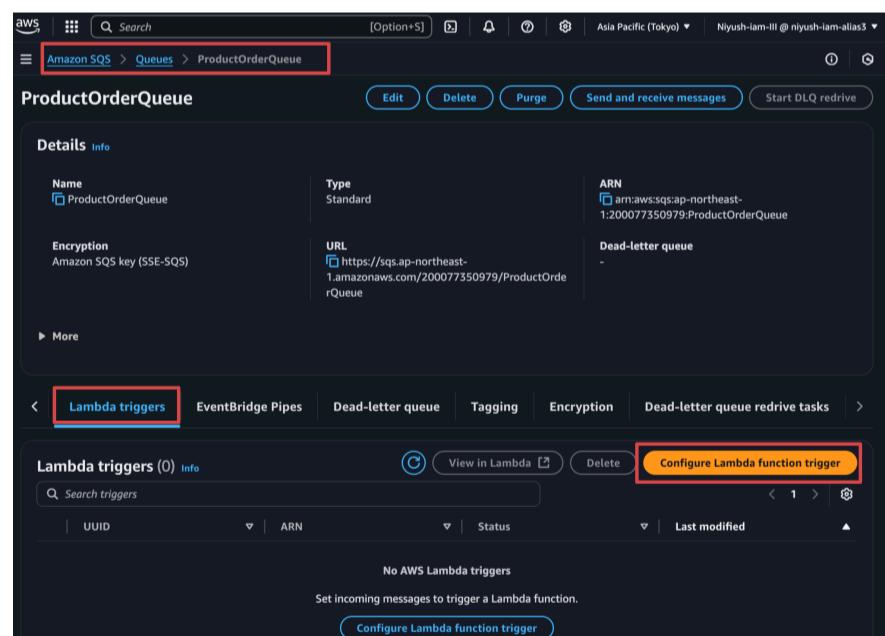




## Step 4 : Deploy and test the application.

- Go to SQS console and configure lambda trigger and specify the [ProcessOrderFunction](#) .
- Check the DynamoDB table to see if the first test event was processed.
- Test the code using the cloudshell.  
- create a file name [input.json](#)

▼ [Image] Configure lambda trigger

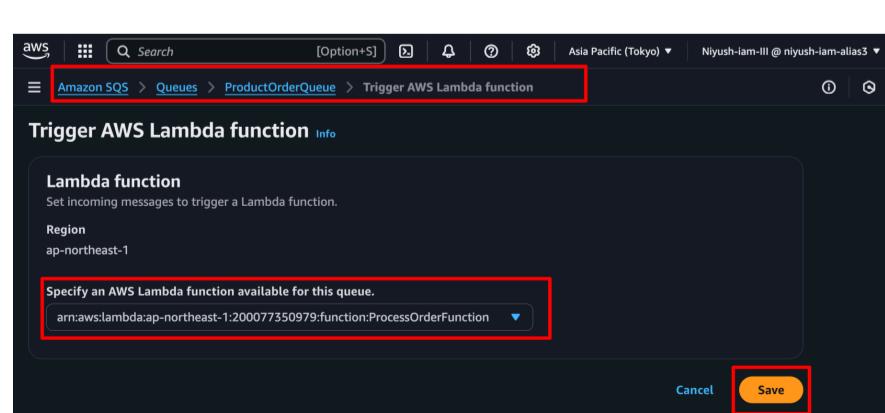


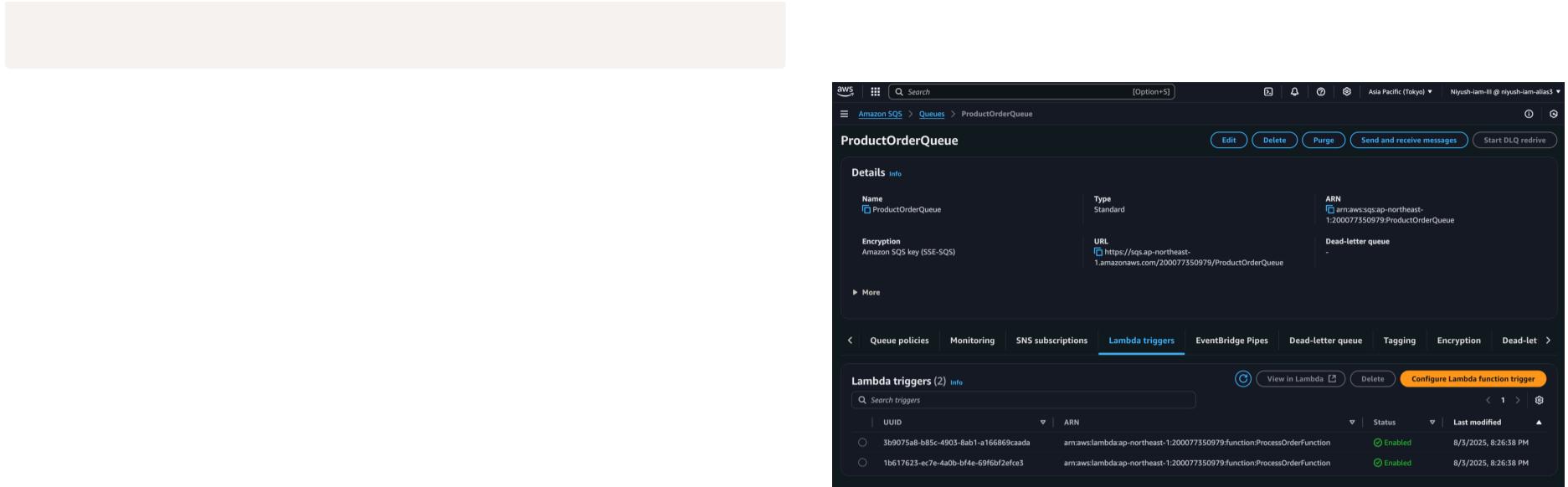
```
pwd
ls
touch input.json
nano input.json
{
  "body": "{\"productName\":\"Test Product 2\",\"quantity\":2}"
}
```

ctrl +z → Y ⇒ enter

- Invoke the lambda function using the following command.

```
aws lambda invoke --function-name SubmitOrderFunction --payload fileb://input.json output.json
```





▼ [Image] input.json file

```

~ $ pwd
/home/cloudshell-user
~ $ ls
~ $ touch input.json
~ $ ls
input.json
~ $ nano input.json
~ $ ls
input.json
~ $ cat input.json
{
  "body": "{\"productName\":\"Test Product 2\",\"quantity\":2}"
}
~ $ 

```

▼ [Image] invoke the function.

```

ap-northeast-1 | +
~ $ aws lambda invoke --function-name SubmitOrderFunction --payload file://input.json output.json
{
  "StatusCode": 200,
  "ExecutedVersion": "$LATEST"
}
~ $ 

```

▼ [Image] check the invoked data in dynamoDB

DynamoDB > Explore Items > ProductOrdersTable

ProductOrdersTable

Tables (1)

Scan or query items

Completed Items returned: 1 - Items scanned: 1 - Efficiency: 100% - RCU consumed: 2

orderId (String)	orderDate (String)	productNa... (String)	quantity (Number)
43bdaf52-3fb4-4be1-...	2025-08-03T11:38:42.600Z	Test Product 2	2

## Step 5 : Creating the REST API

- Go to API Gateway → Create API → REST API  
API name : ProductOrdersAPI
- create a new resource name : orders  
enable : CORS
- create a POST method for /orders integrated with the SubmitOrderFunction
- Enter a Lambda proxy integration.
- Click back up to the /orders resource and click "enable CORS"
- Select all CORS options.
- Deploy API to a new stage named prod
- Update the invoke URL in index.html

▼ [Image] Creating the API

API Gateway > APIs > Create API

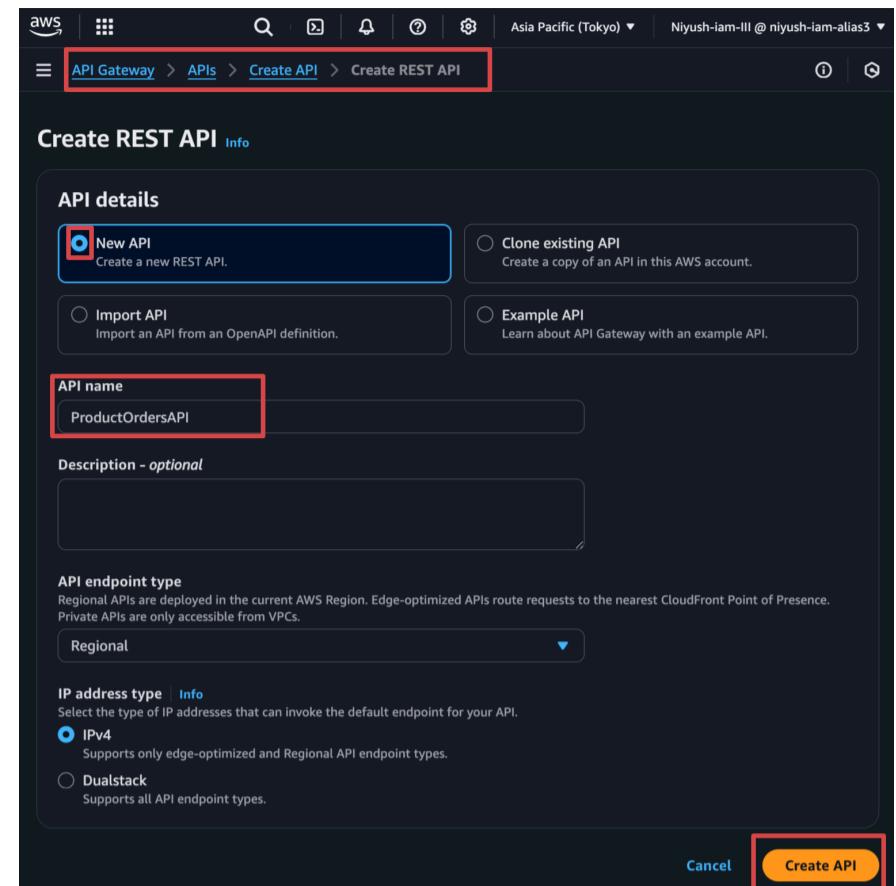
WebSocket API

Build a WebSocket API using persistent connections for real-time use cases such as chat applications or dashboards.

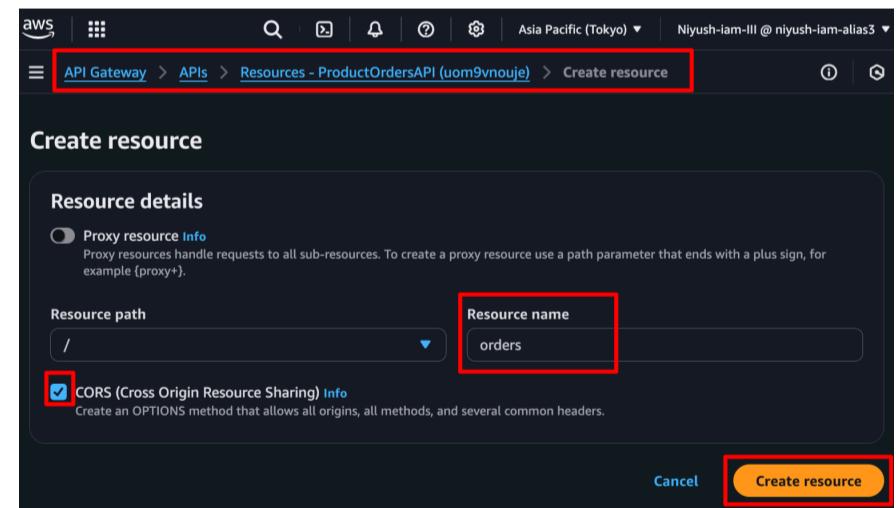
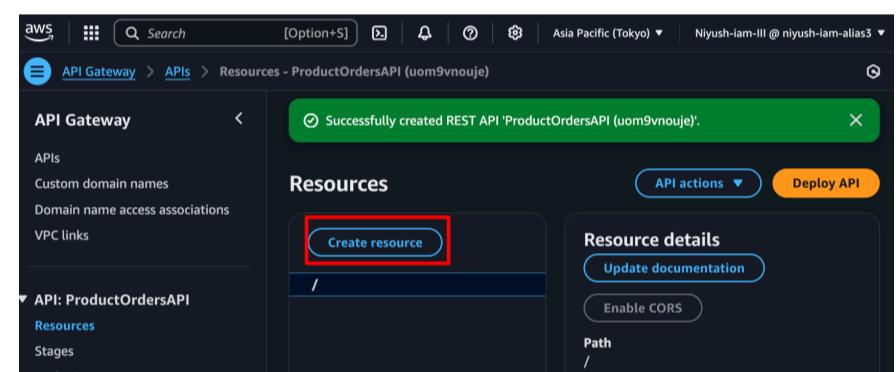
REST API

Develop a REST API where you gain complete control over the request and response along with API management capabilities.

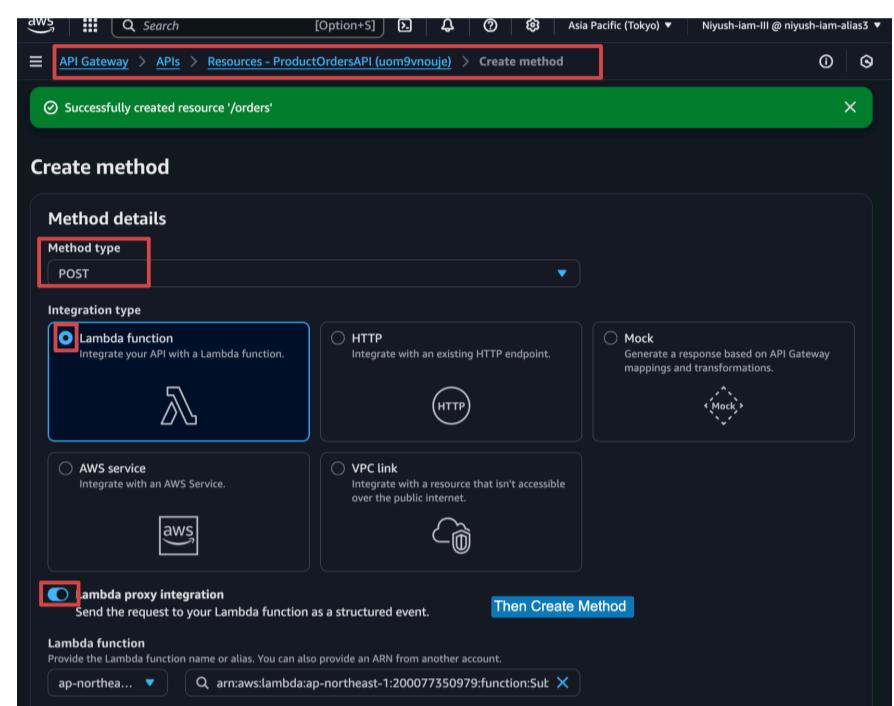
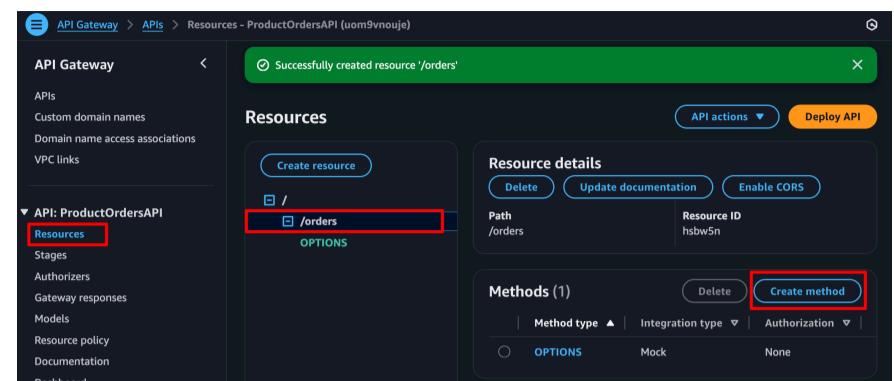
Import Build



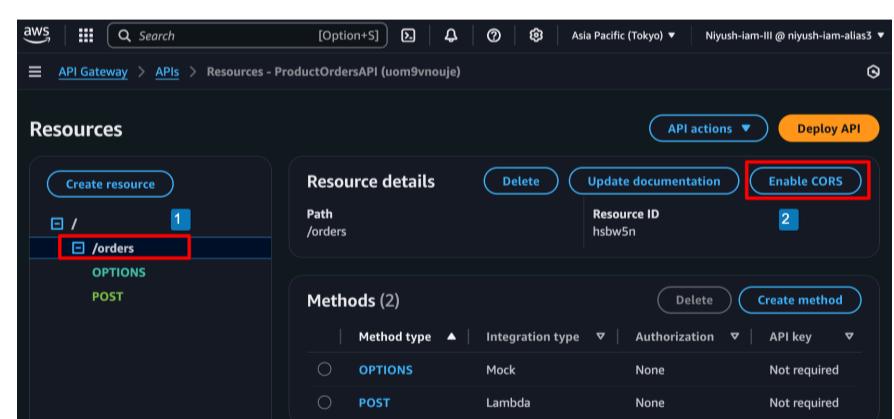
▼ [Image] Creating Resources

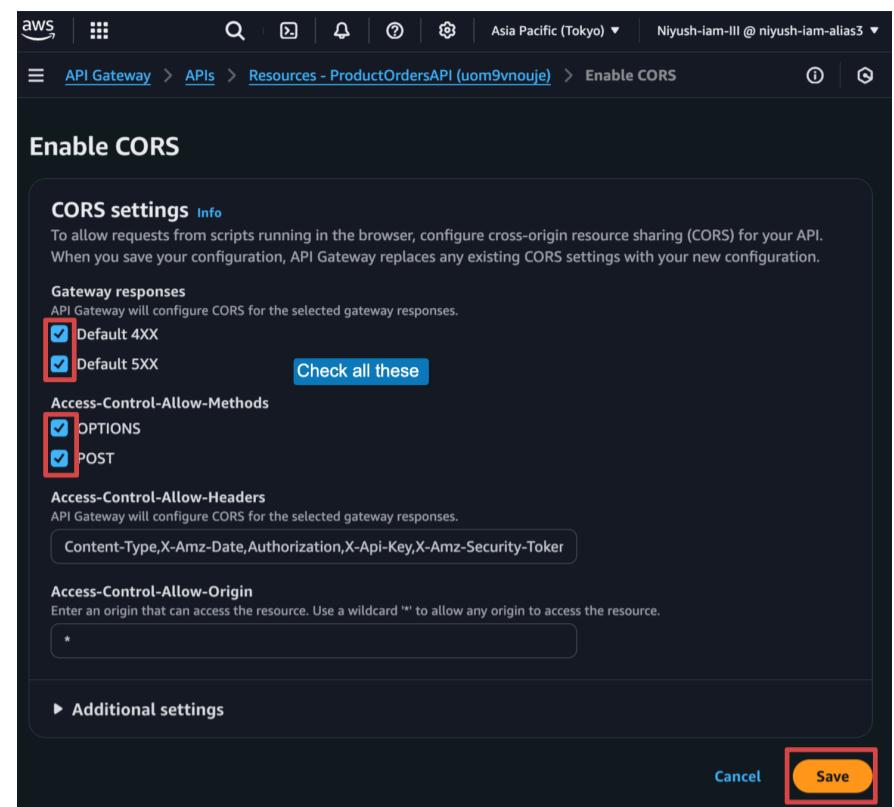


▼ [Image] Creating POST method

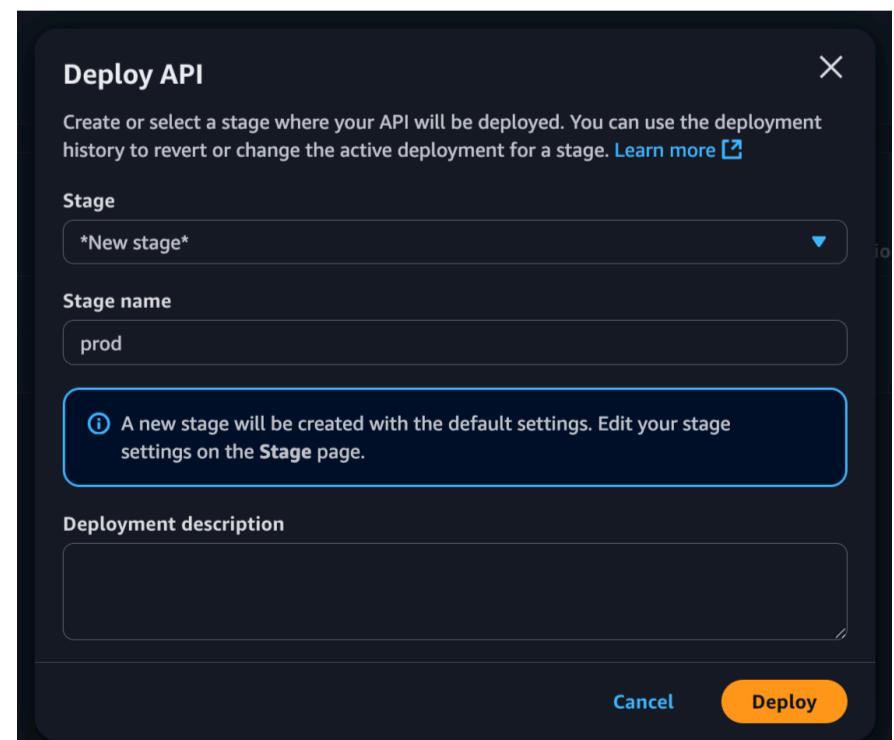
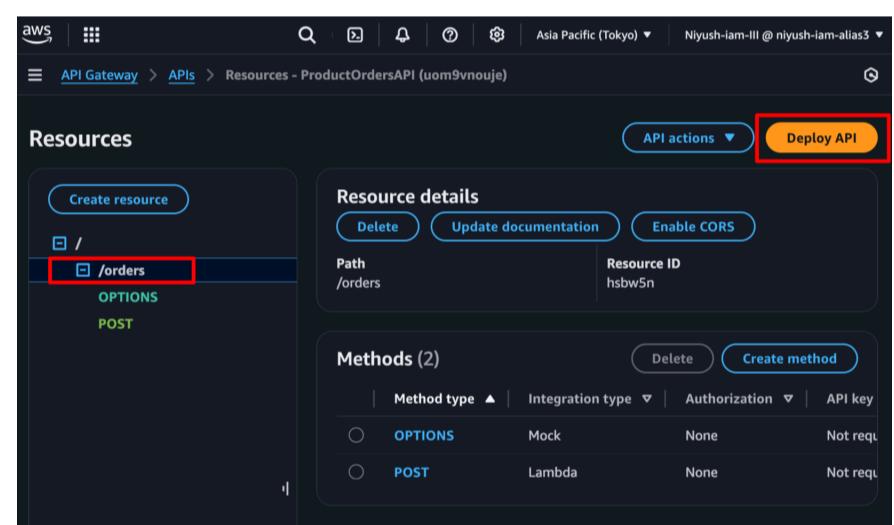


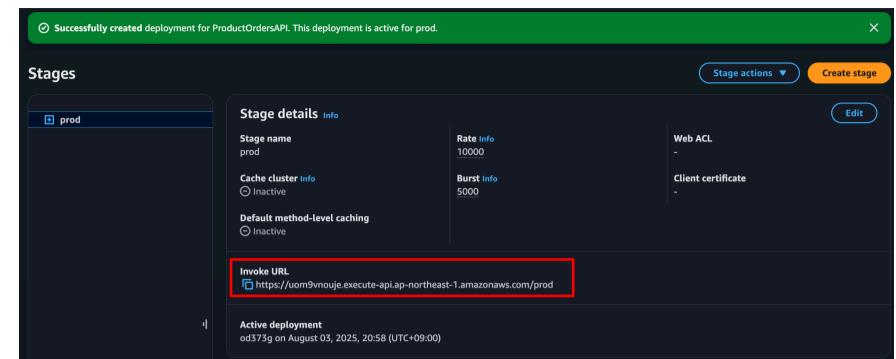
▼ [Image] Enabling CORS





▼ [Image] Deploy API and update the index.html





▼ [Image] Update the API of index.html

```

index.html U
index.html > html > body > div.container > form#orderForm
  2   <html lang="en">
  17  <body>
  18    <div class="container">
  20      <form id="orderForm">
  22        <input type="text" id="productName" name="productName">
  23        <label for="quantity">Quantity:</label>
  24        <input type="number" id="quantity" name="quantity">
  25        <input type="button" value="Submit" onclick="submitOrder()">
  26      </form>
  27    </div>
  28    <script>
  29      async function submitOrder() {
  30        const productName = document.getElementById('productName').value;
  31        const quantity = document.getElementById('quantity').value;
  32        const response = await fetch('https://uom9vnouje.execute-api.ap-northeast-1.amazonaws.com/prod/orders', {
  33          method: 'POST',
  34          body: JSON.stringify({ productName, quantity }),
  35          headers: { 'Content-Type': 'application/json' }
  36        });
  37        if (response.ok) { alert('Order submitted successfully!'); document.getElementById('orderForm').reset(); }
  38        else { alert('Failed to submit order!'); }
  39      }
  40    </script>
  41  </body>
  42</html>

```

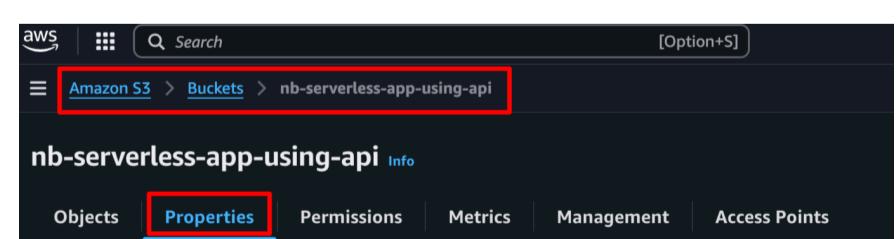
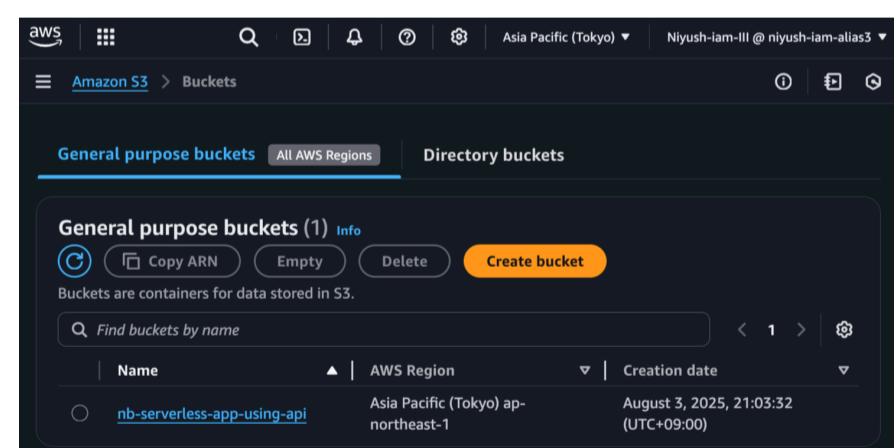
## Step 6 : Create the S3 bucket

- Go to the S3 bucket console.
- Name the bucket
- Configure the bucket for static website for hosting.
- Make sure to set default document to `index.html`
- updated `index.html`
- Enable public access using a bucket policy.

▼ The required policy to allow public access.

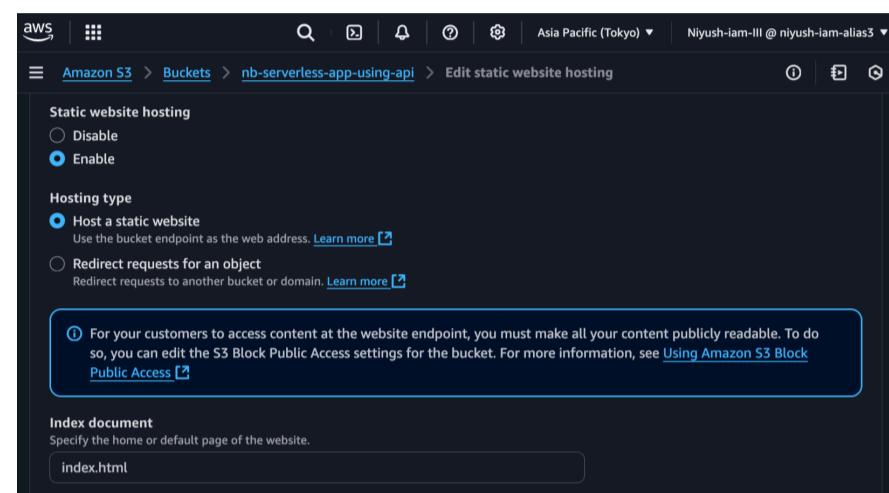
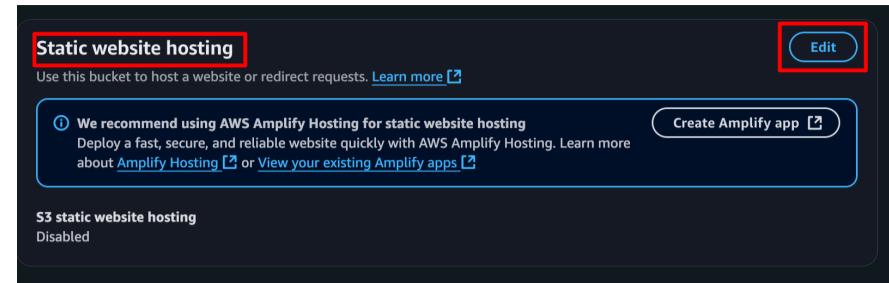
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "PublicReadGetObject",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "<YOUR-BUCKET-ARN>/*"
    }
  ]
}
```

▼ [Image] Creating bucket and configure settings

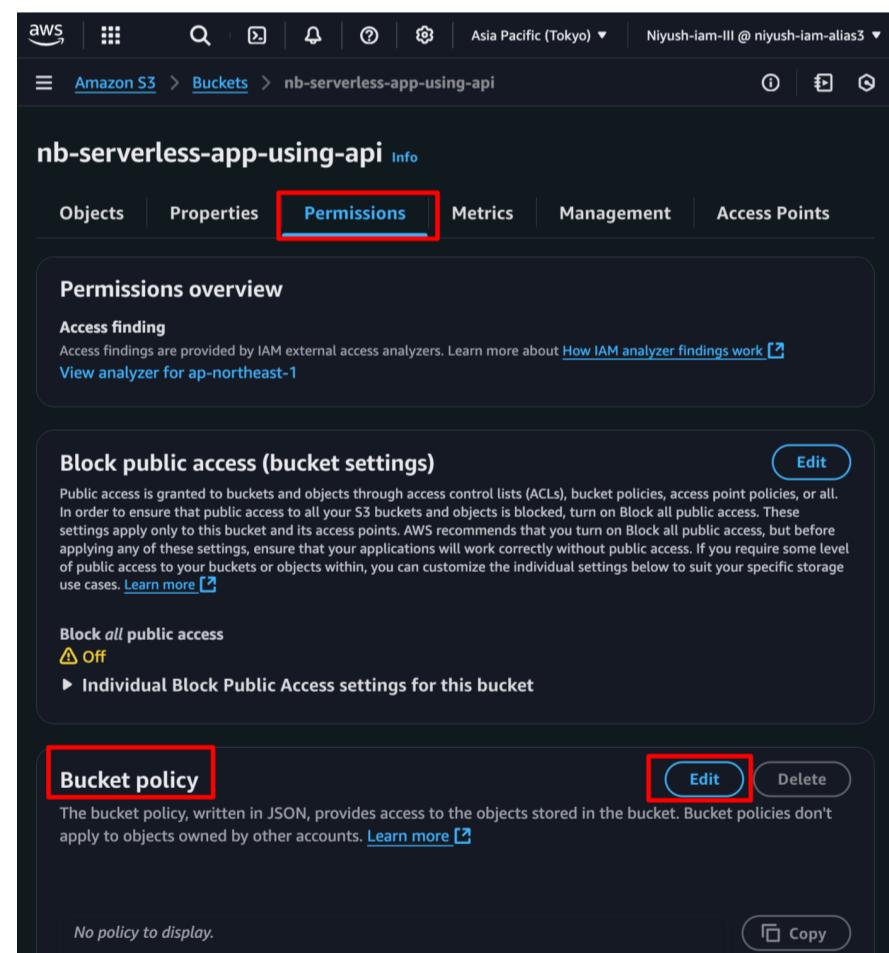




- once all done finally upload the updated index.html
- Once the upload is completed head to **properites** → **static website hosting** → click on **Bucket website endpoint**



▼ [Image] adding permission and editing bucket policy



```

1 v {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Sid": "PublicReadGetObject",
6       "Effect": "Allow",
7       "Principal": "*",
8       "Action": "s3:GetObject",
9       "Resource": "arn:aws:s3:::nb-serverless-app-using-api/*"
10    }
11  ]
12 }

```

The screenshot shows the AWS S3 Bucket Policy editor. A JSON policy document is displayed, allowing public read access to all objects in the bucket. The 'Save changes' button at the bottom right is highlighted with a red box.

▼ [Image] upload the index.html

The screenshot shows the AWS S3 Bucket Objects page for 'nb-serverless-app-using-api'. The 'Upload' button in the top navigation bar is highlighted with a red box. The 'Actions' dropdown menu is also highlighted with a red box.

The screenshot shows the AWS S3 Bucket Upload page. The 'Add files' button in the 'Files and folders (0)' section is highlighted with a red box. The 'Add folder' button is also visible.

**Files and folders (1 total, 2.1 KB)**

[Remove](#) [Add files](#) [Add folder](#)

All files and folders in this table will be uploaded.

	Name	Folder	Type	Size
<input type="checkbox"/>	index.html	-	text/html	2.1 KB

**Destination** [Info](#)

**Destination**  
[s3://nb-serverless-app-using-api](#)

▶ **Destination details**  
Bucket settings that impact new objects stored in the specified destination.

▶ **Permissions**  
Grant public access and access to other AWS accounts.

▶ **Properties**  
Specify storage class, encryption settings, tags, and more.

[Cancel](#) [Upload](#)

▼ [Image] Accessing the static website

aws | Search [Option+S]

Amazon S3 > Buckets > nb-serverless-app-using-api

**nb-serverless-app-using-api** [Info](#)

Objects Properties Permissions Metrics Management Access Points

**Static website hosting**

Use this bucket to host a website or redirect requests. [Learn more](#)

We recommend using AWS Amplify Hosting for static website hosting  
Deploy a fast, secure, and reliable website quickly with AWS Amplify Hosting. Learn more about [Amplify Hosting](#) or [View your existing Amplify apps](#)

S3 static website hosting  
Enabled

Hosting type  
Bucket hosting

**Bucket website endpoint**  
When you configure your bucket as a static website, the website is available at the AWS Region-specific website endpoint of the bucket. [Learn more](#)  
<http://nb-serverless-app-using-api.s3-website-ap-northeast-1.amazonaws.com>

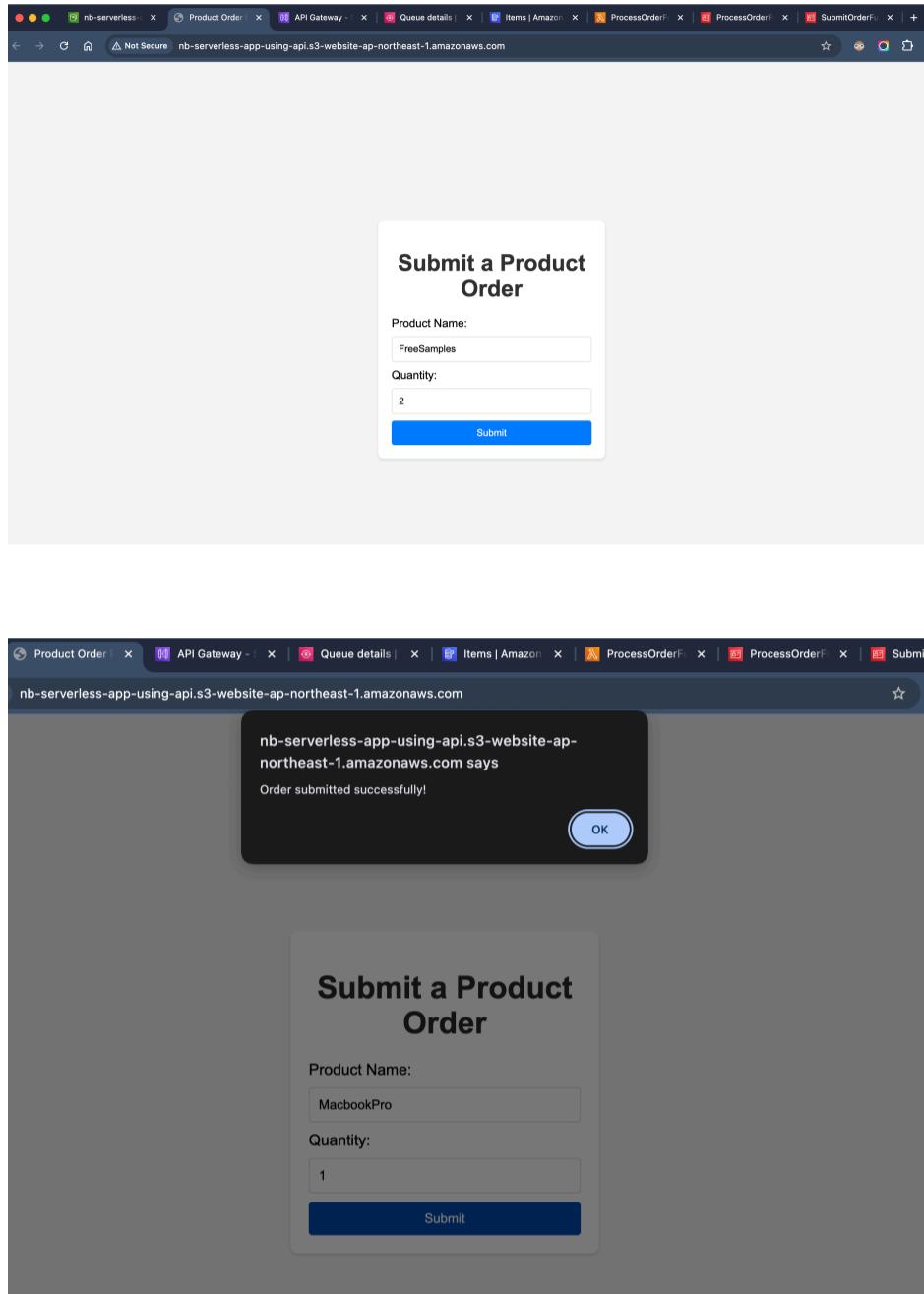
nb-serverless... Product Order API Gateway Queue details Items Amazon ProcessOrder ProcessOrder SubmitOrder

Not Secure nb-serverless-app-using-api.s3-website-ap-northeast-1.amazonaws.com

**Submit a Product Order**

Product Name:   
Quantity:   
[Submit](#)

## Step 7 : Testing time



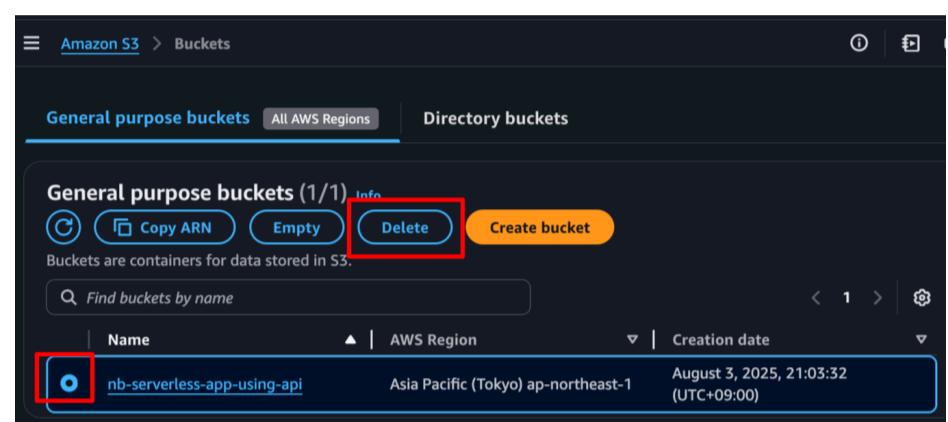
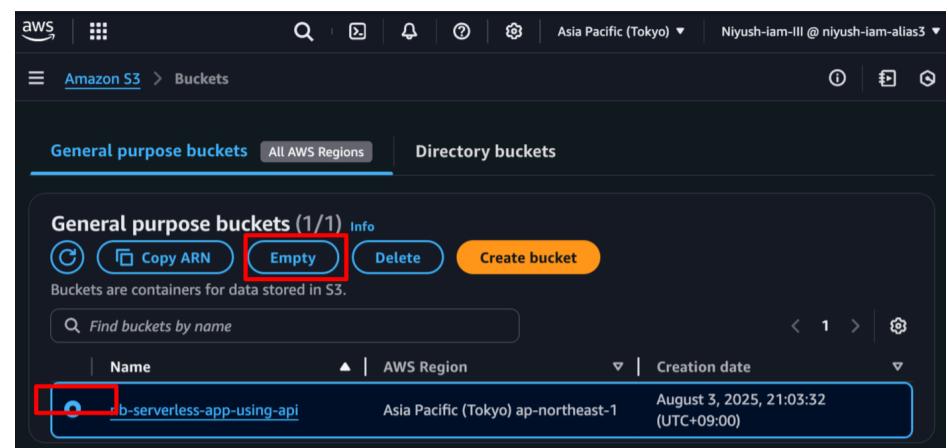
- Head to DynamoDb and check if the orders are successfully stored or not

▼ [Image] Check the orders in DynamoDB

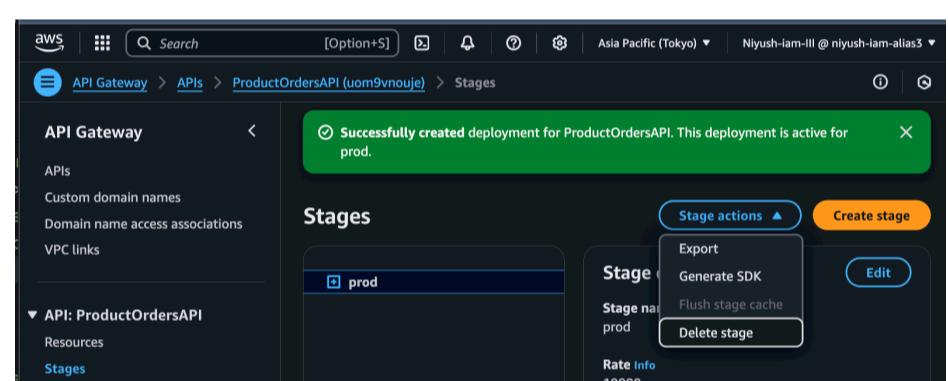
A screenshot of the AWS DynamoDB console. The left sidebar shows 'DynamoDB' with 'Explore items' selected. The main area shows the 'ProductOrdersTable' with a scan configuration. The results table displays three items:

orderId (String)	orderDate	productName	quantity
43b8af52-3fbd-4be1-8151-1cd9a29a8c0	2025-08-03T11:38:42.600Z	Test Product 2	2
846aee0a-7fe5-4acf-9707-85ea451e5825	2025-08-03T12:18:06.981Z	MacbookPro	1
f93039e0-4c94-405a-8a8f-9ef34c6a9c8d	2025-08-03T12:17:40.292Z	FreeSamples	2

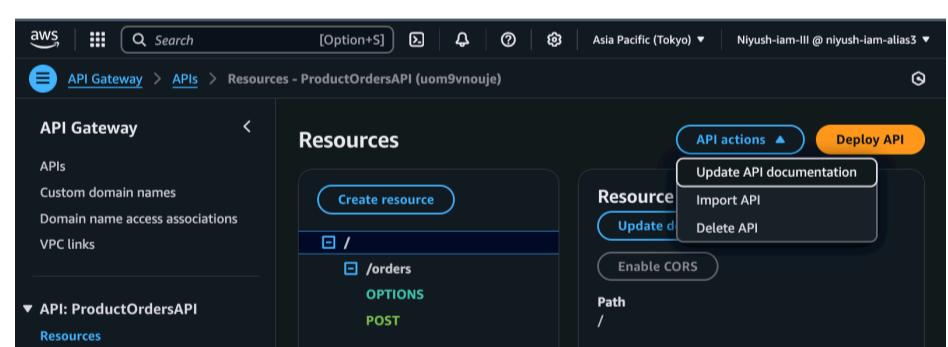
## Step 8 : Delete the resources



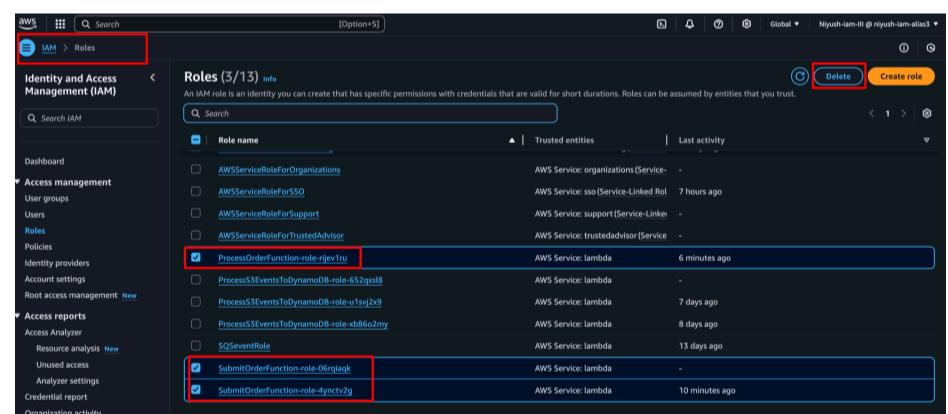
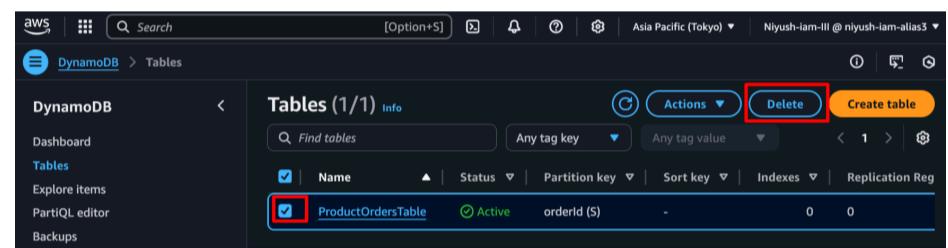
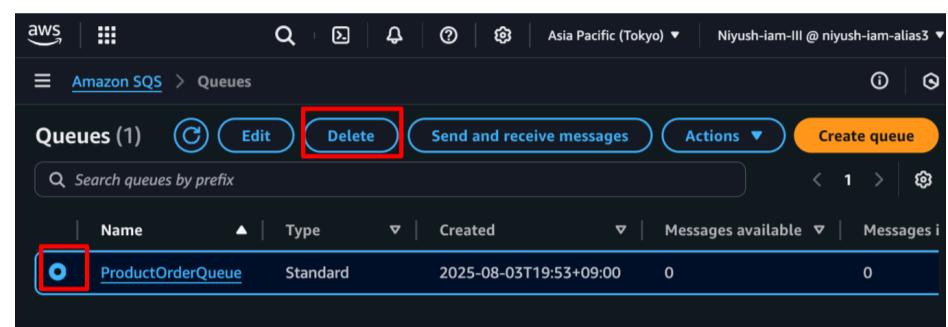
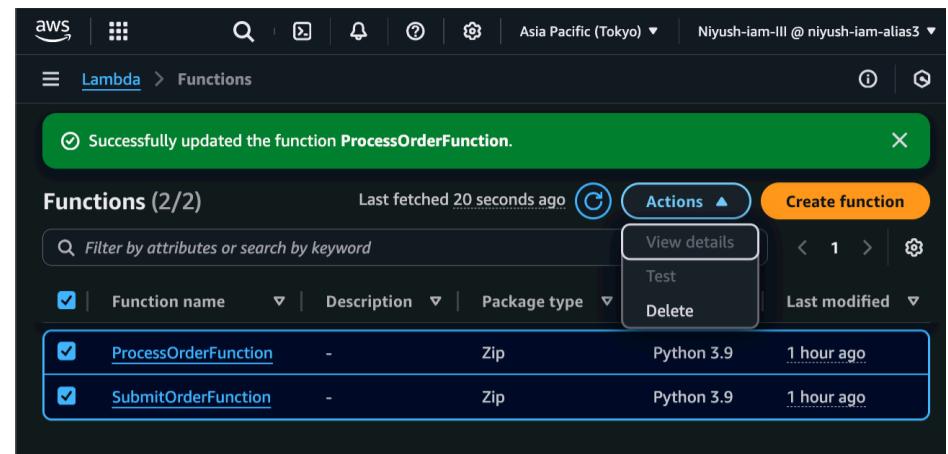
deleting the stage



deleting the resources



deleting lambda functions.



And that's a wrap!!!!!!!!!

niyush

buy me some caffeine

<http://buymeacoffee.com/niyushbjr1L>

