Serverless Labs

1. Building a Serverless Web Application

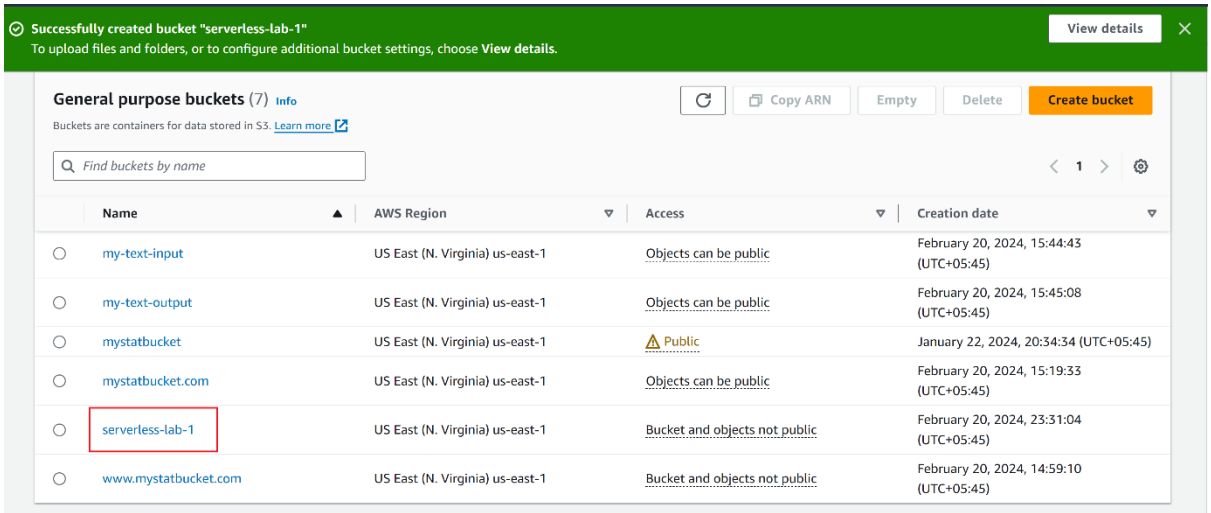
Objective: Create a serverless web application using AWS Lambda, API Gateway, S3, and DynamoDB.

Approach:

- **Set Up Backend**: Create Lambda functions to handle backend logic. These functions will interact with a DynamoDB table for data storage.
- **API Gateway**: Set up API Gateway to create RESTful endpoints that trigger the Lambda functions.
- **Frontend Hosting**: Host a static website on S3 that interacts with the backend via API Gateway.
- **Integration**: Ensure that the frontend can successfully send requests to the backend and display responses.

Goal: Understand the basics of building and connecting serverless backend services with a static frontend, enabling a fully serverless web application.

First, let's initiate the process by creating an S3 bucket with the required details and policies. After the bucket creation, we'll proceed to upload the static webpage file and activate static hosting by setting its property to 'true.'



From here, we will access our bucket > go to edit bucket policy, and open policy generator.

Amazon S3 > Buckets > serverless-lab-1 > Edit bucket policy

Edit bucket policy Info

Bucket policy

Policy examples | Policy generator

The bucket policy, written in JSON, provides access to the objects stored in the bucket. Bucket policies don't apply to objects owned by other accounts. Learn more

Bucket ARN

arn:aws:s3:::serverless-lab-1

Policy

amazon web services

AWS Policy Generator

The AWS Policy Generator is a tool that enables you to create policies that control access to Amazon Web Services (AWS) products and resources. For more information about creating policies, see key concepts in Using AWS Identity and Access Management. Here are sample policies.

Step 1: Select Policy Type

A Policy is a container for permissions. The different types of policies you can create are an IAM Policy, an S3 Bucket Policy, an SNS Topic Policy, a VPC Endpoint Policy, and an SQS Queue Policy.

Select Type of Policy S3 Bucket Policy

Step 2: Add Statement(s)

A statement is the formal description of a single permission. See a description of elements that you can use in statements.

Effect Allow Deny

Principal *

Use a comma to separate multiple values.

AWS Service Amazon S3 All Services ('*')

Use multiple statements to add permissions for more than one service.

Actions 1 Action(s) Selected All Actions ('*')

Amazon Resource Name (ARN) arn:aws:s3:::serverless-lab-1

ARN should follow the following format: arn:aws:s3:::${BucketName}/${KeyName}.
Use a comma to separate multiple values.

Add Conditions (Optional)

Add Statement

Step 3: Generate Policy

A *policy* is a document (written in the Access Policy Language) that acts as a container for one or more statements.

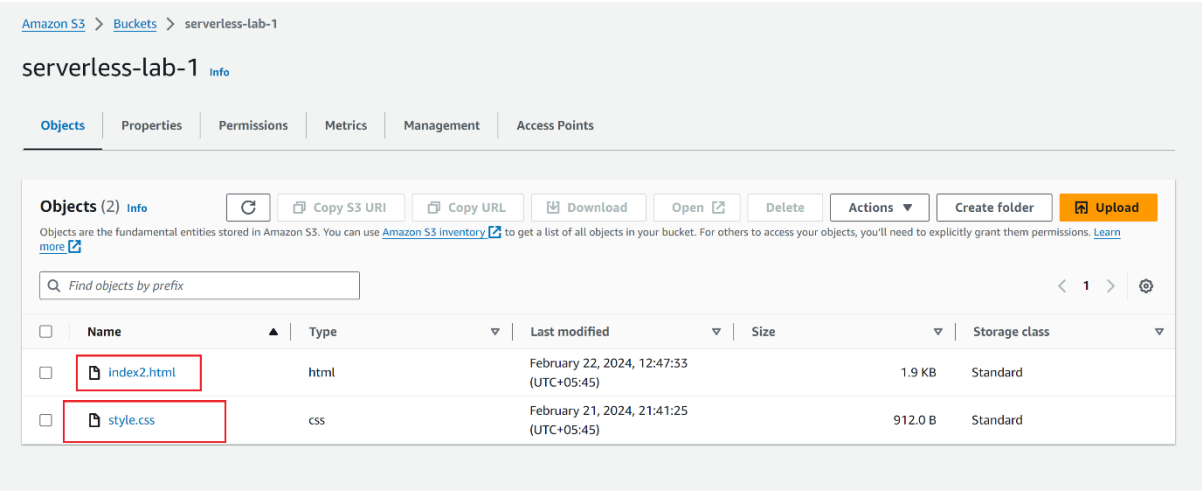
Add one or more statements above to generate a policy.

Policy JSON Document

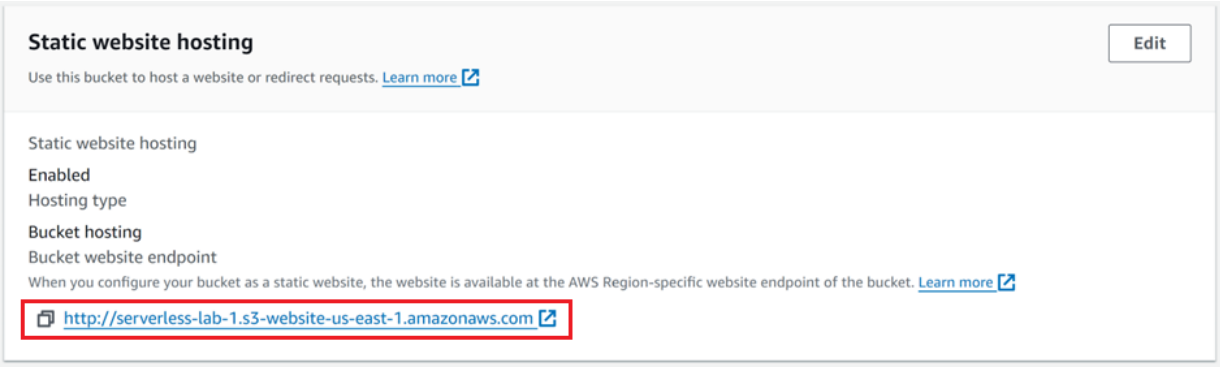
Click below to edit. To save the policy, copy the text below to a text editor.
Changes made below will not be reflected in the policy generator tool.

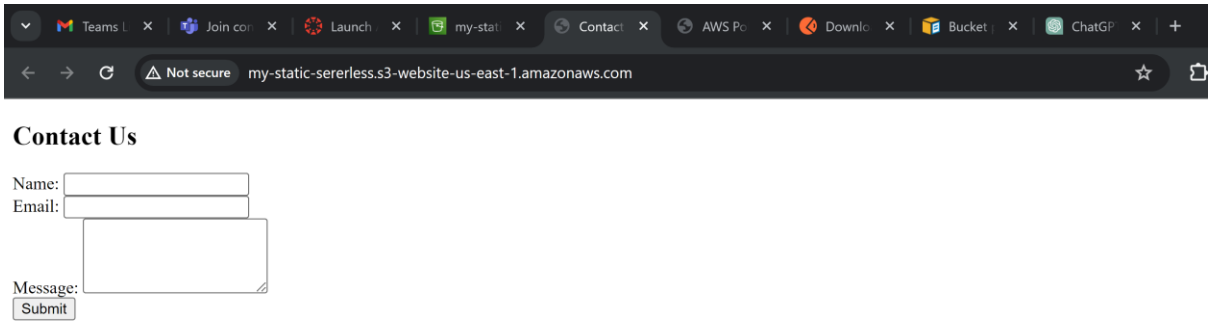
```
{
  "Id": "Policy1708581371646",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1708581368439",
      "Action": [
        "s3:GetObject"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws:s3:::serverless-lab-1",
      "Principal": "*"
    }
  ]
}
```


Close

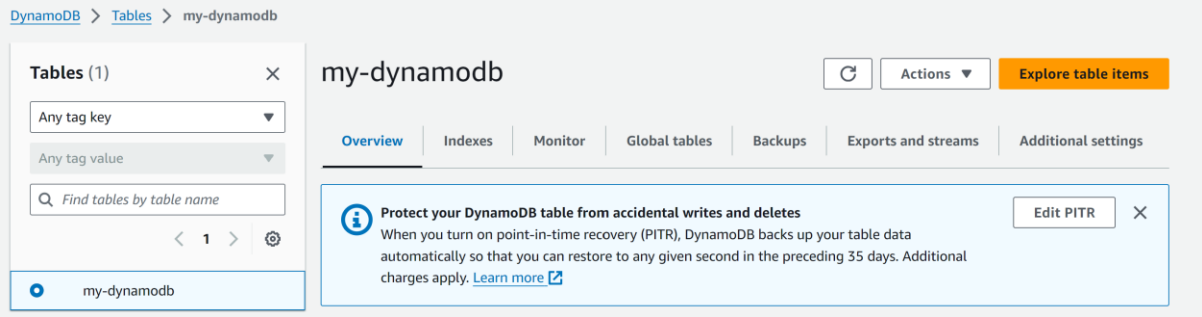


We can access our static website from the following link:

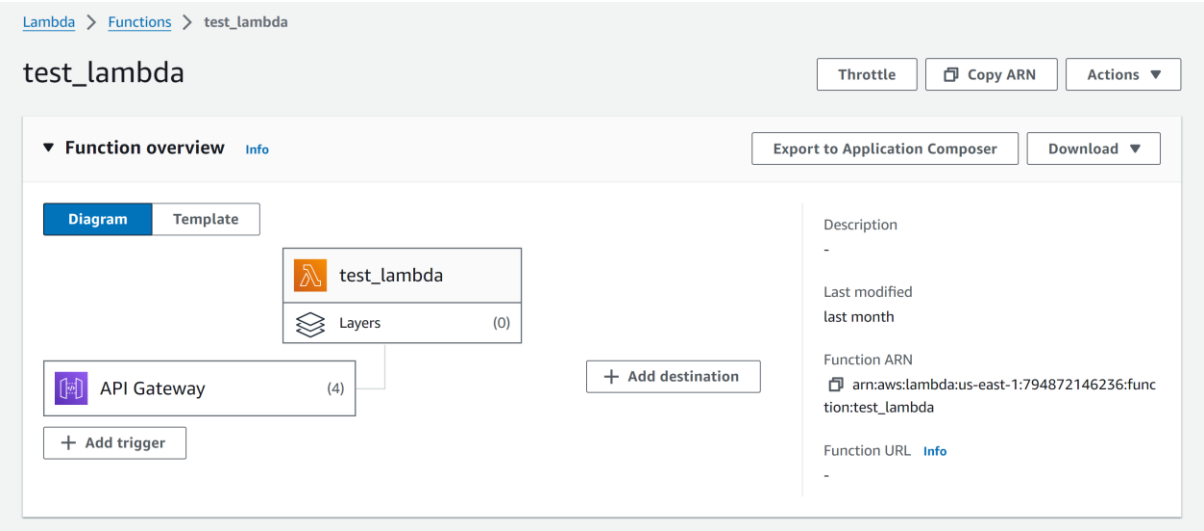




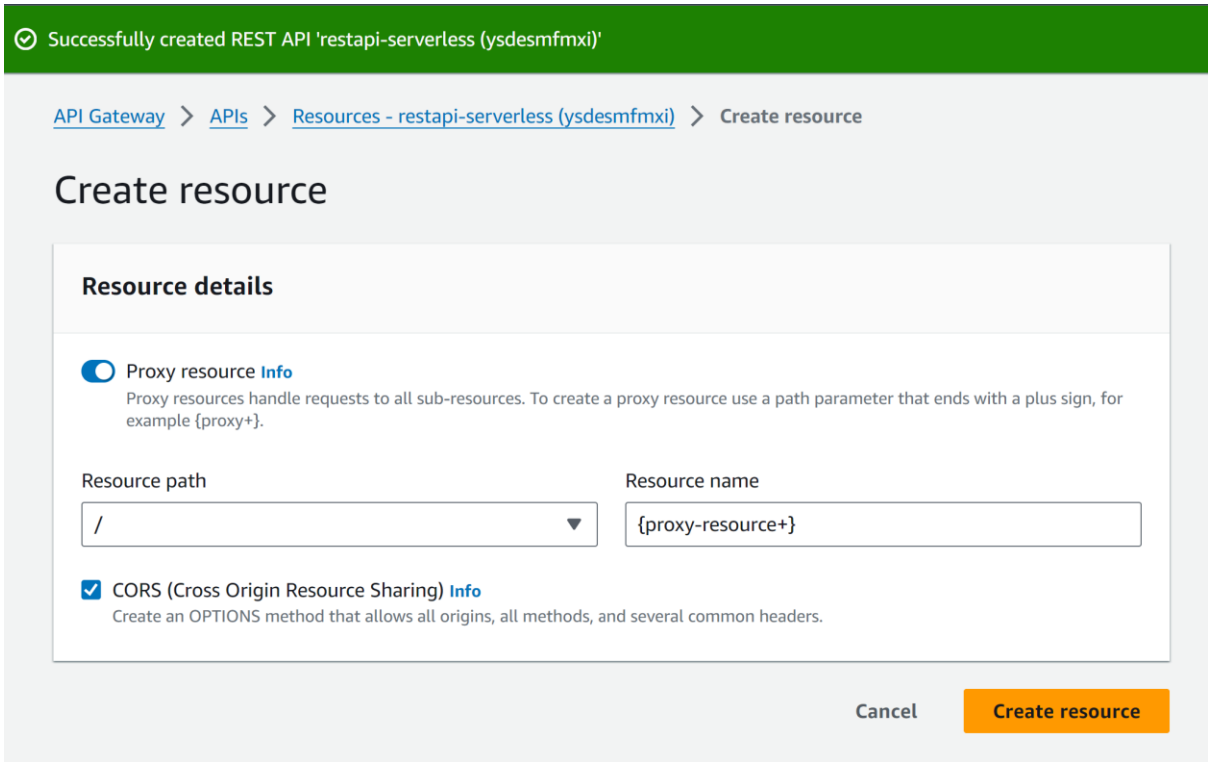
Next, we will create a table in DynamoDB with necessary configurations.



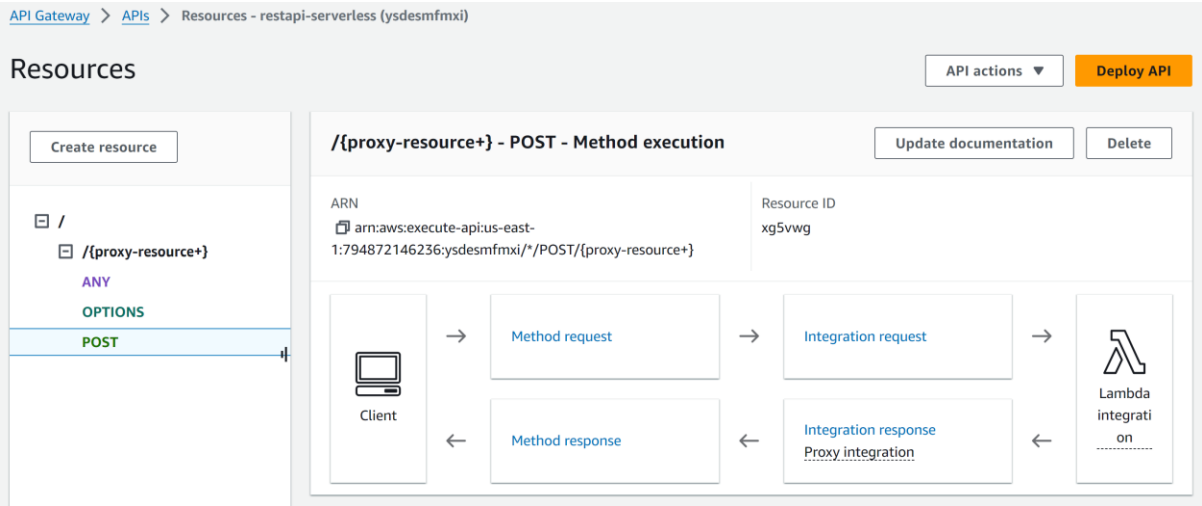
Next, we will create lambda function with API Gateway and necessary configuration

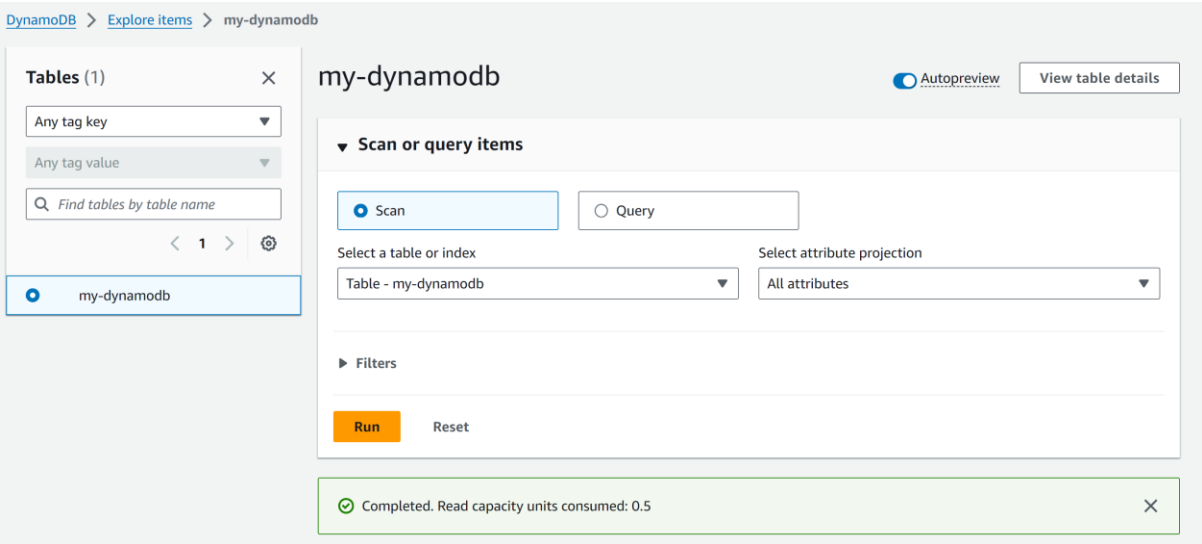


Next, let's create a REST API with a resource. When configuring the resource, ensure to enable the proxy resource. By default, the resource is set to non-proxy integration, which means it only retrieves the request payload data in the event and not other details like headers.

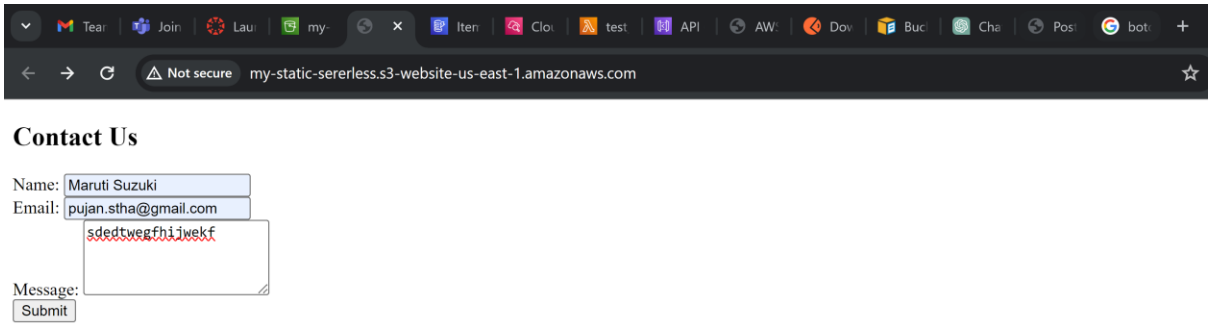


We will now create a POST function and deploy the API

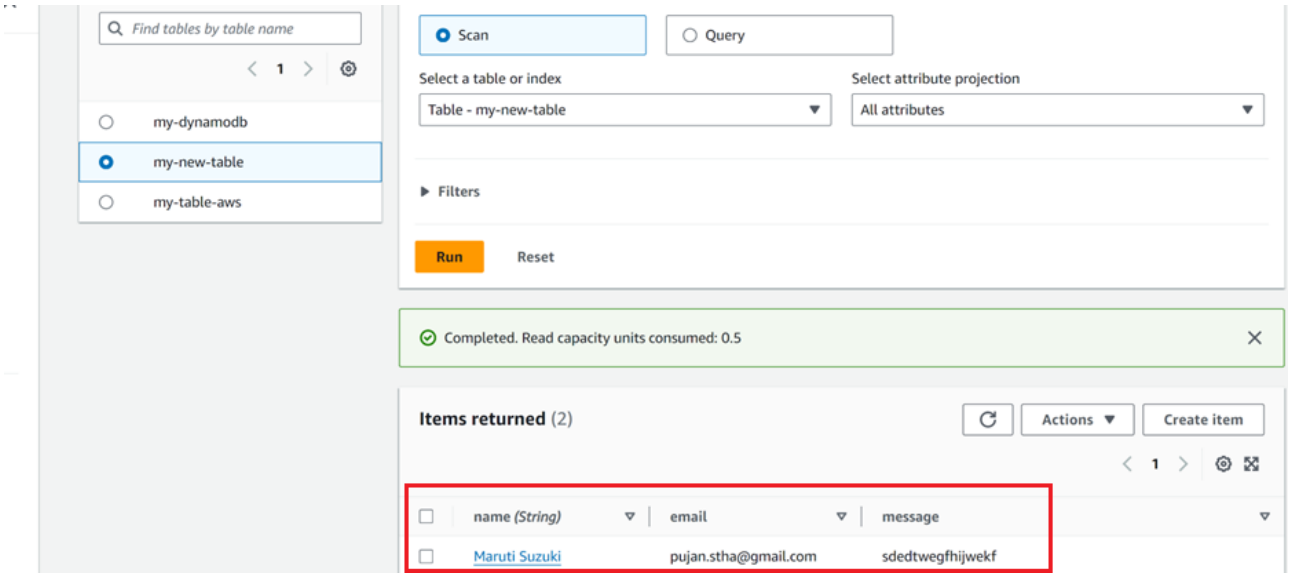




After this, we will go back to our static website and fill the forms.



The form POST when submitted, can be seen below.



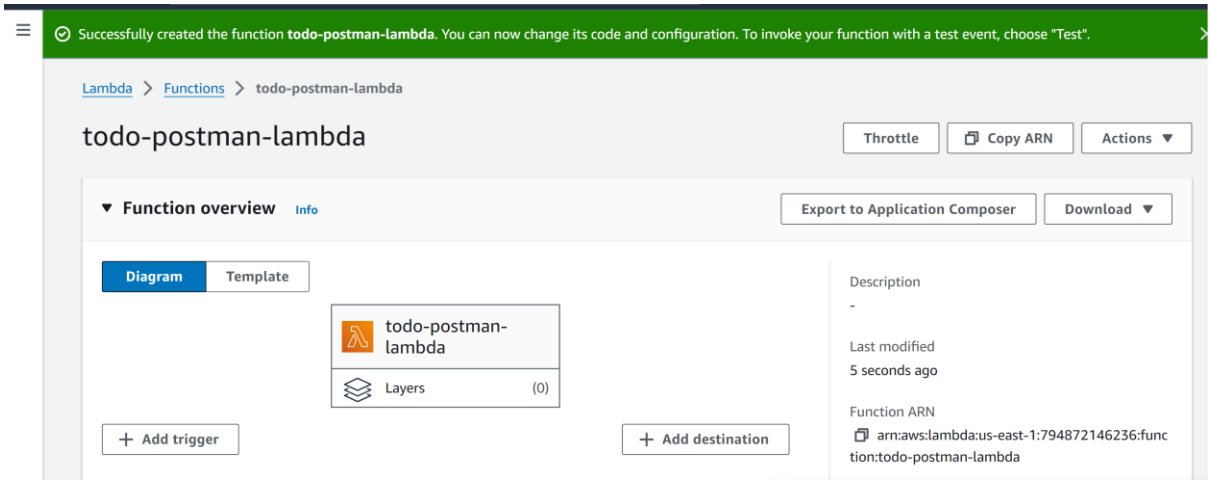
2. Creating a Serverless API

Objective: Develop a serverless API using AWS Lambda and API Gateway.

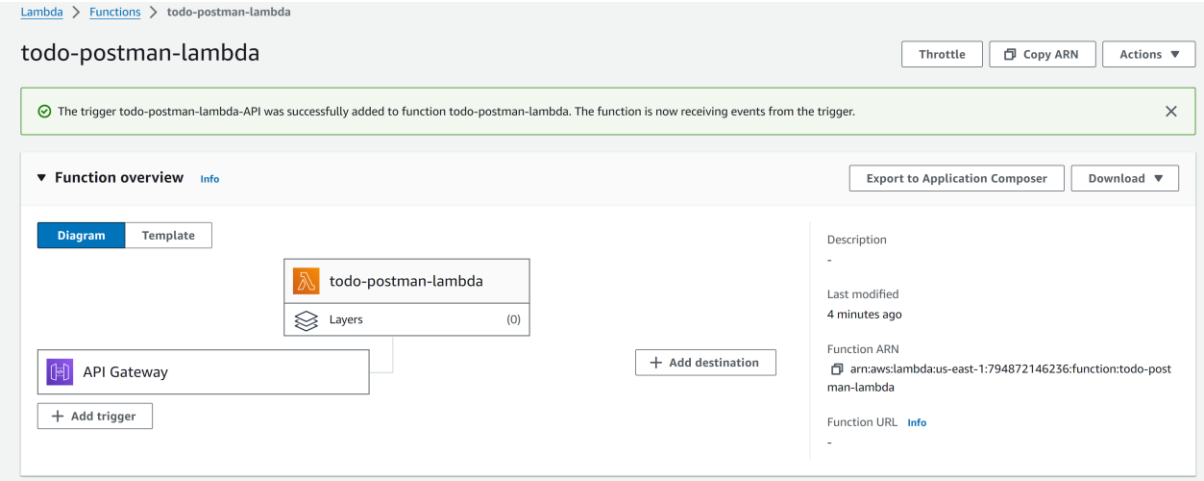
Approach:

- **Define API**: Design a simple RESTful API (e.g., for a todo list application).
- **Lambda Functions**: Create Lambda functions for each API method (GET, POST, PUT, DELETE).
- **API Gateway Setup**: Use API Gateway to set up the API endpoints, connecting each endpoint to the corresponding Lambda function.
- **Testing**: Test the API using tools like Postman or AWS API Gateway test functionality.

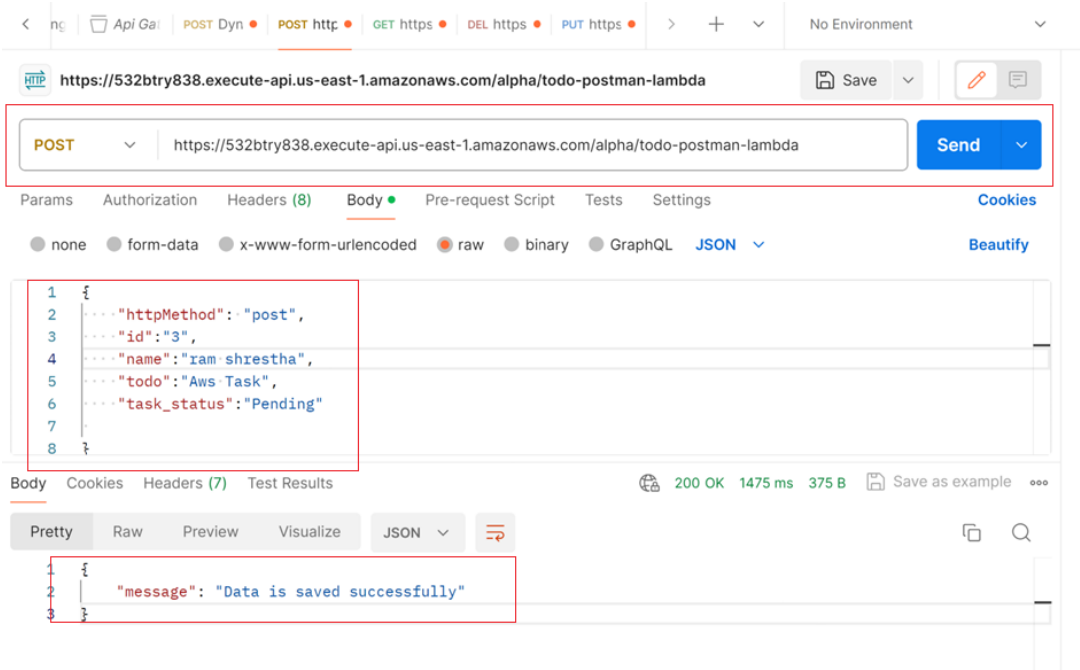
Goal: Gain hands-on experience in building and deploying a serverless API, understanding the integration between Lambda and API Gateway



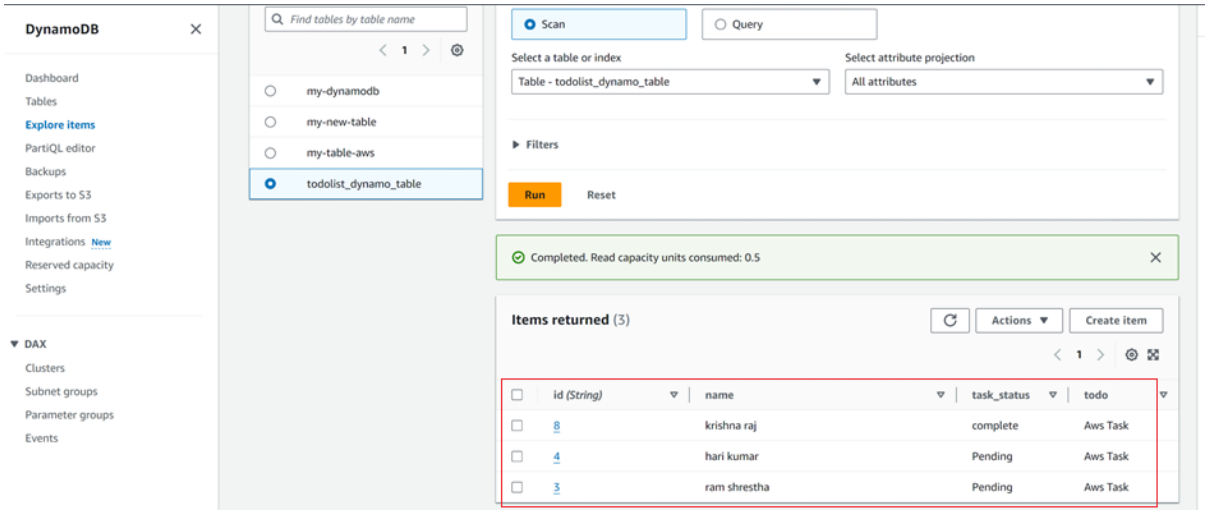
Create API Gateway



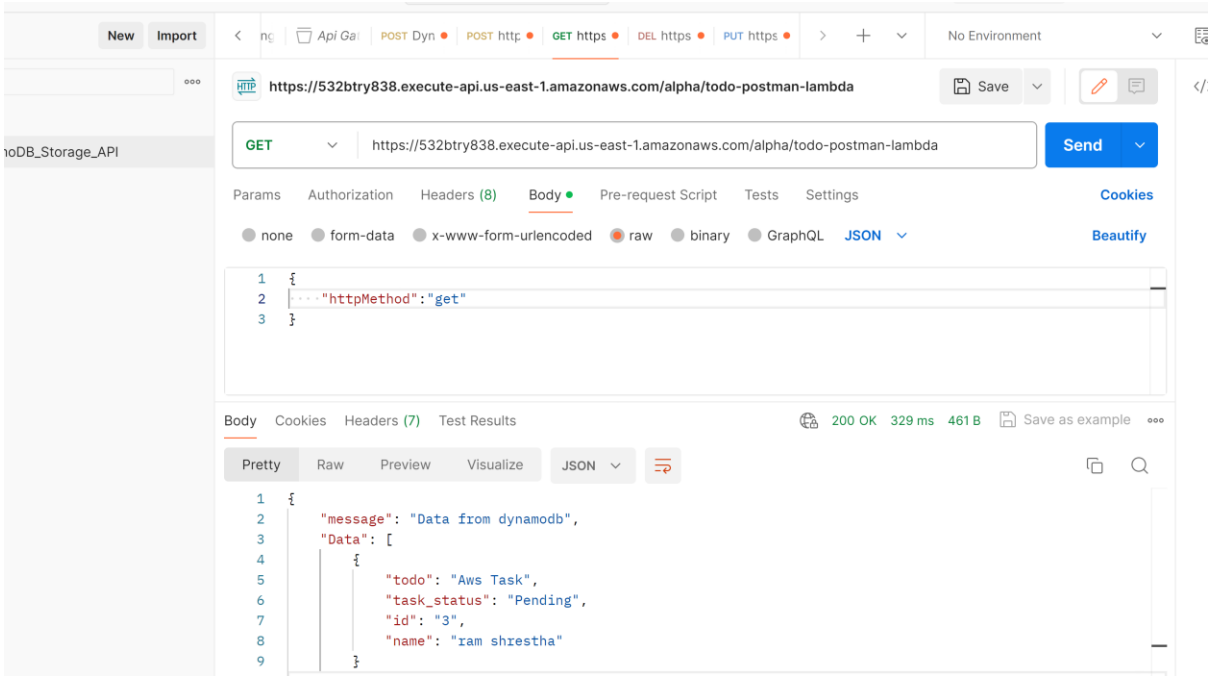
POST METHOD:

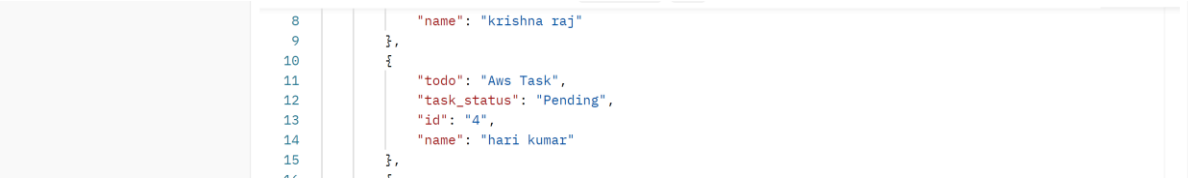


Now when we check in the DynamoDB:

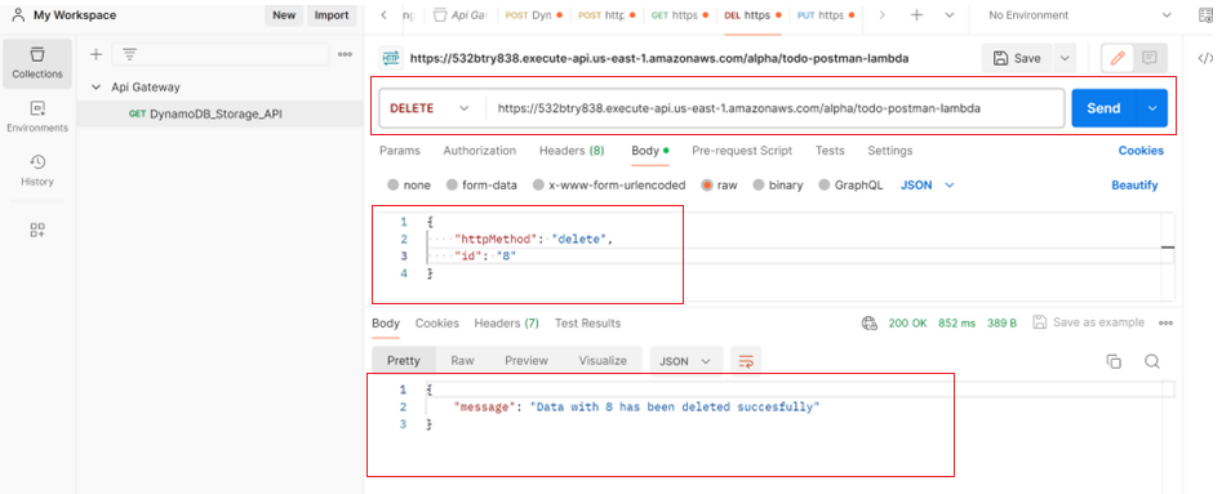


Now we will check GET METHOD:

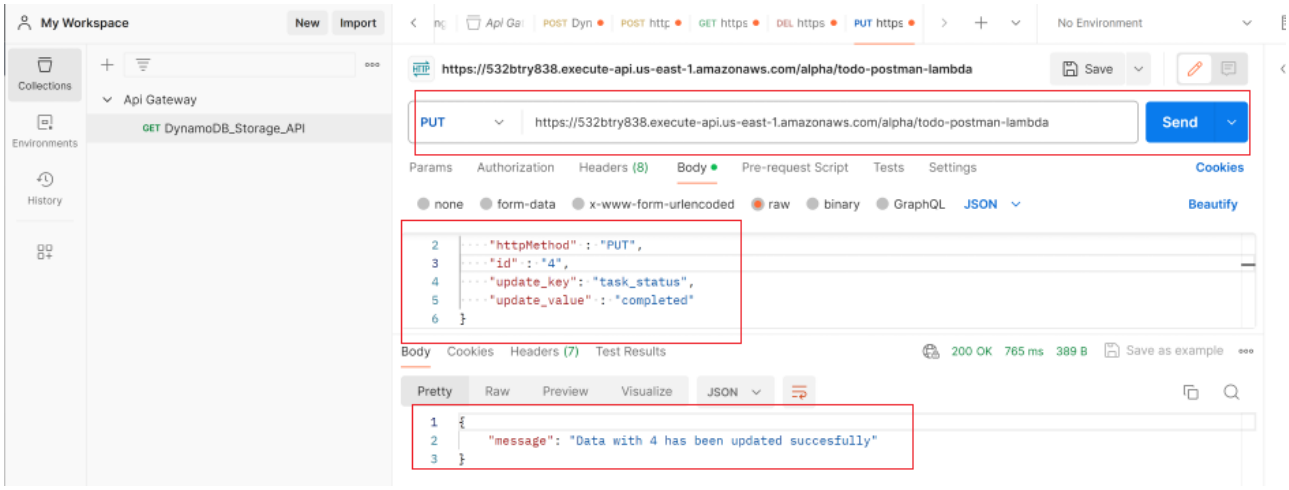




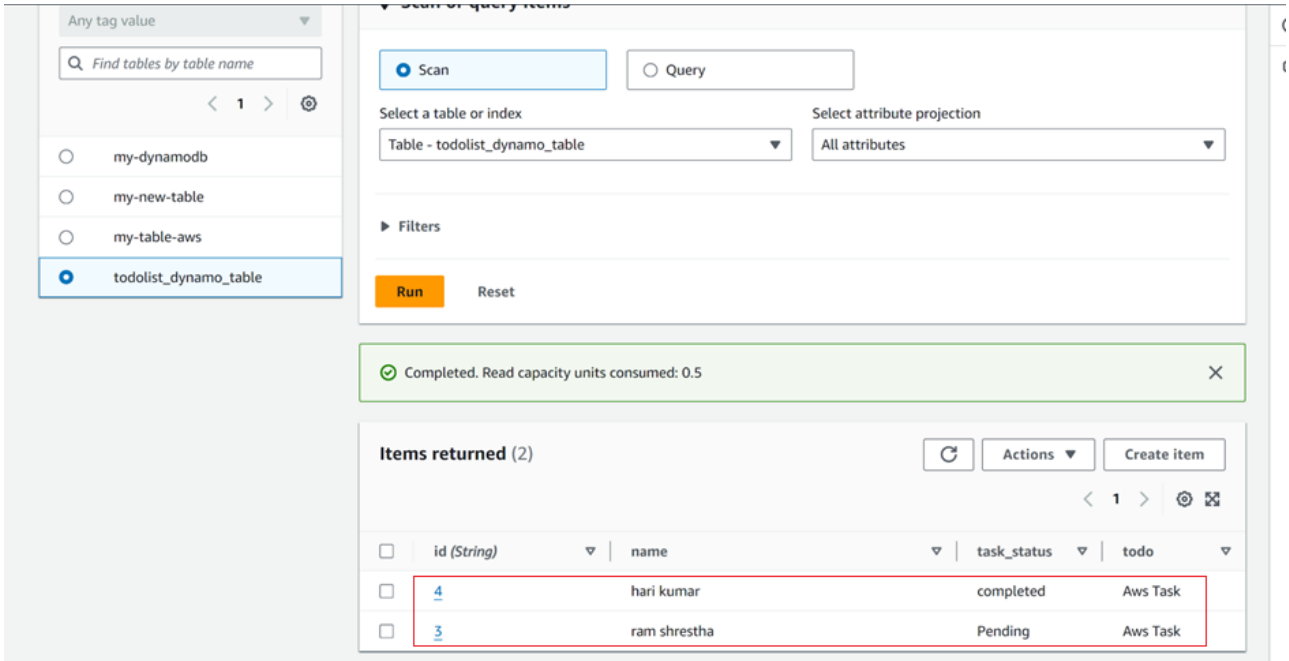
Now we will check DELETE METHOD:



Now we will use UPDATE(PUT) METHOD:



Now when we check our database:



3. Serverless Data Processing Pipeline

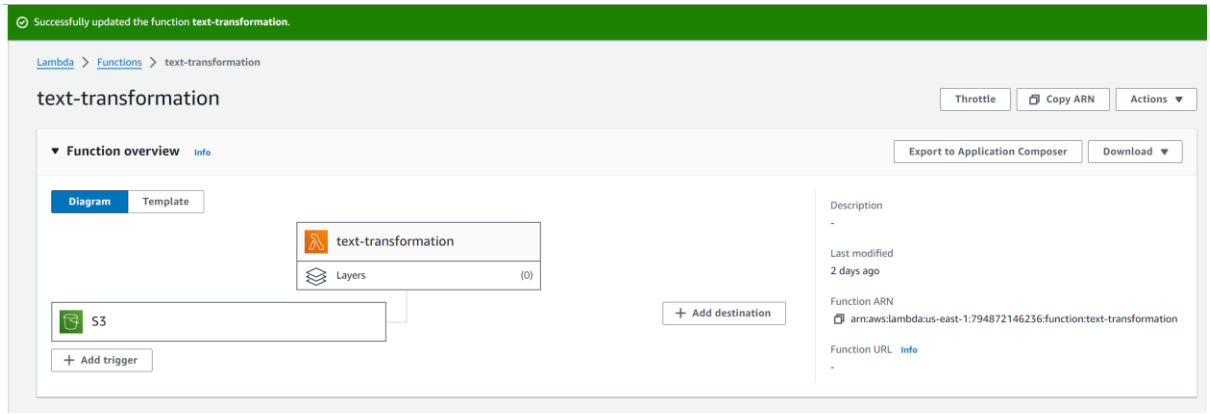
Objective: Build a serverless pipeline for processing data (e.g., log processing or ETL jobs).

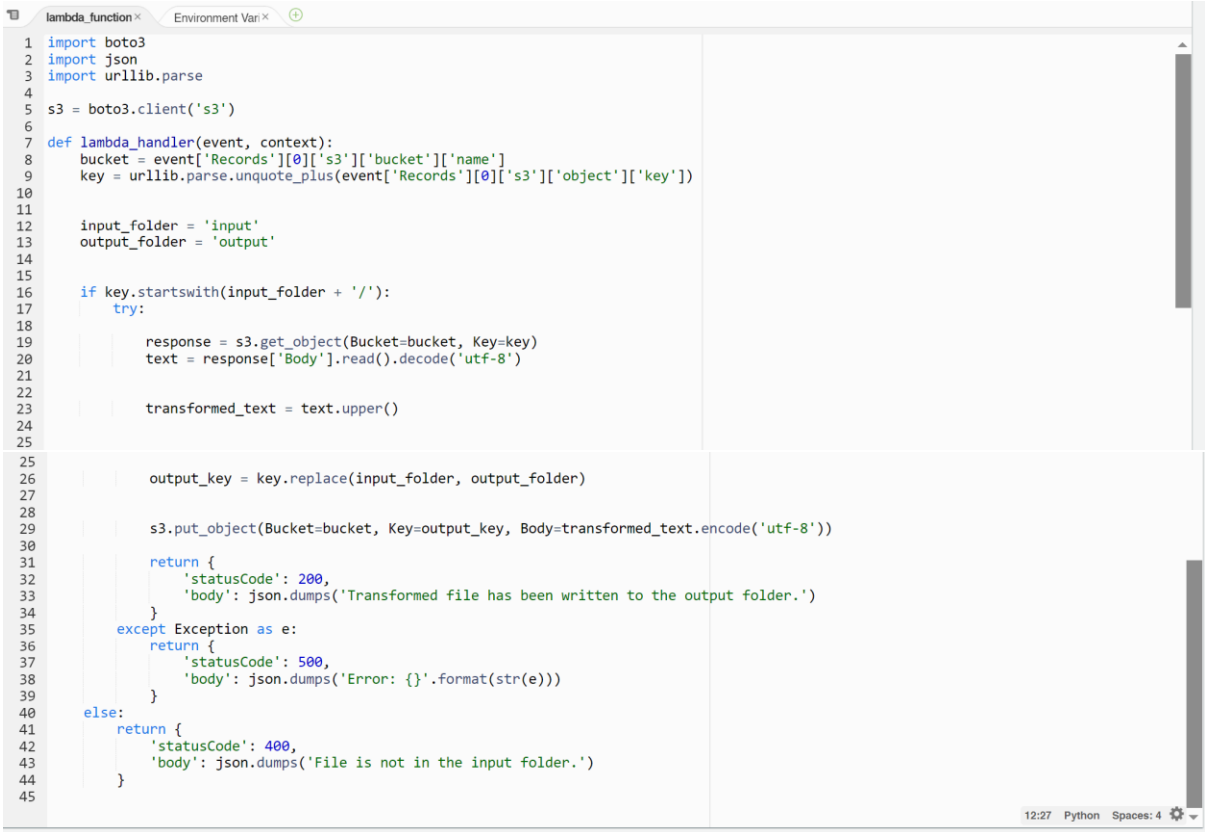
Approach:

- **Data Ingestion**: Use AWS services like S3 or Kinesis to ingest data.
- **Processing**: Create Lambda functions to process the ingested data.
- **Storage**: Store the processed data in an appropriate AWS service, like S3 or DynamoDB.
- **Monitoring**: Set up CloudWatch to monitor the pipeline's performance and to log any issues.

Goal: Learn to build a serverless data processing pipeline, understanding the flow of data through various AWS services.

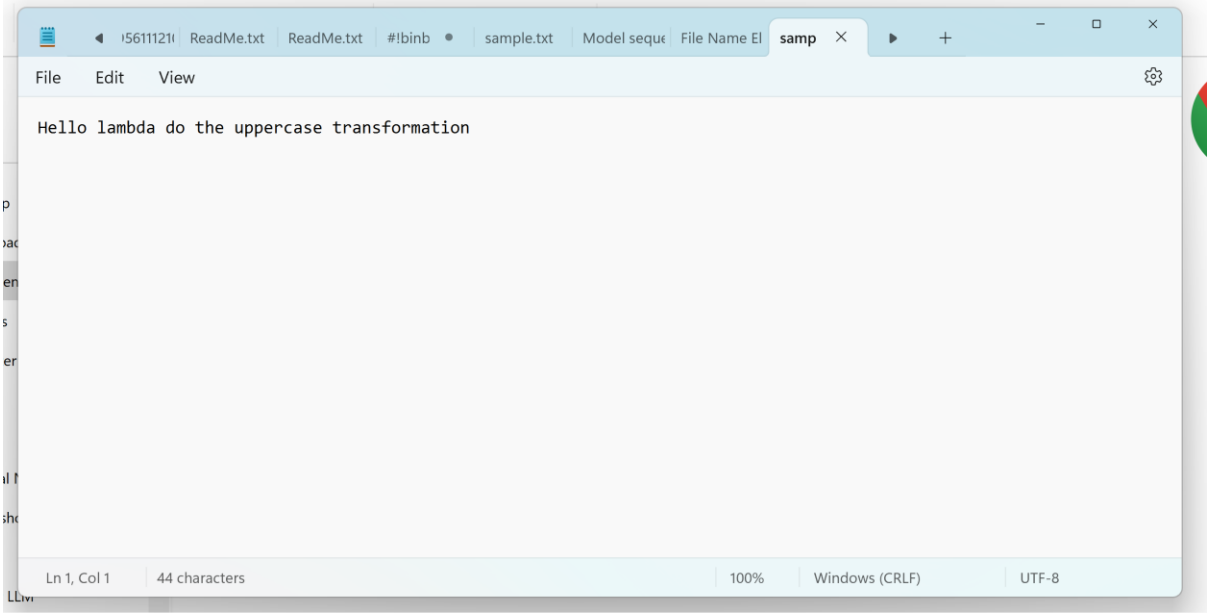
I have created a Lambda function that will trigger S3 bucket for text transformation which will transform the text into Uppercase and store in the S3 bucket inside output folder.



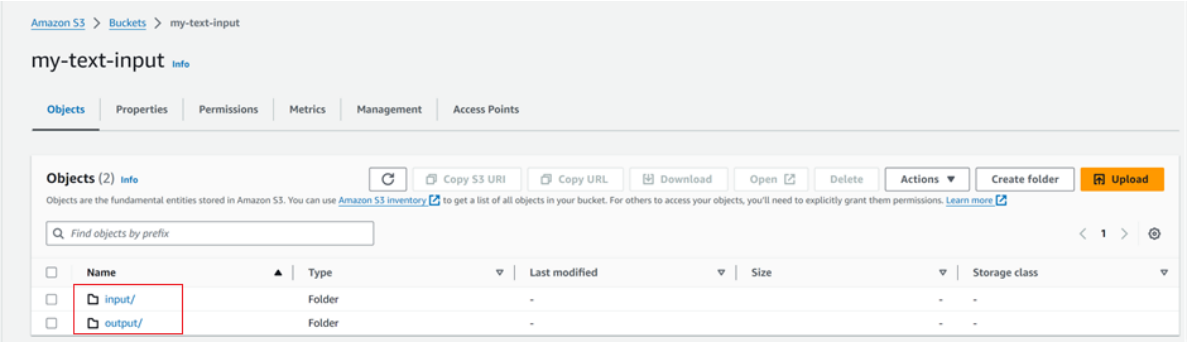


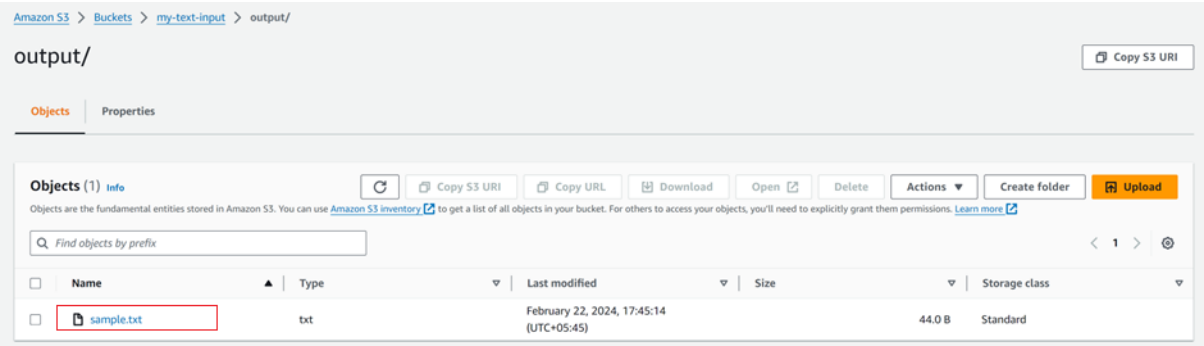

```python
import boto3
import json
import urllib.parse

s3 = boto3.client('s3')

def lambda_handler(event, context):
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'])


    input_folder = 'input'
    output_folder = 'output'


    if key.startswith(input_folder + '/'):
        try:

            response = s3.get_object(Bucket=bucket, Key=key)
            text = response['Body'].read().decode('utf-8')


            transformed_text = text.upper()


            output_key = key.replace(input_folder, output_folder)


            s3.put_object(Bucket=bucket, Key=output_key, Body=transformed_text.encode('utf-8'))

            return {
                'statusCode': 200,
                'body': json.dumps('Transformed file has been written to the output folder.')
            }
        except Exception as e:
            return {
                'statusCode': 500,
                'body': json.dumps('Error: {}'.format(str(e)))
            }
    else:
        return {
            'statusCode': 400,
            'body': json.dumps('File is not in the input folder.')
        }
```

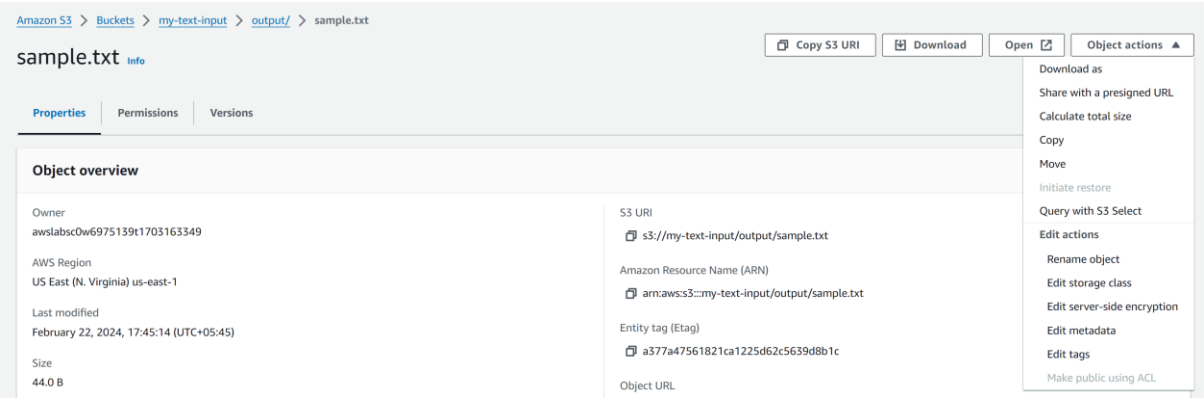
I have a text file in my input folder called sample.txt

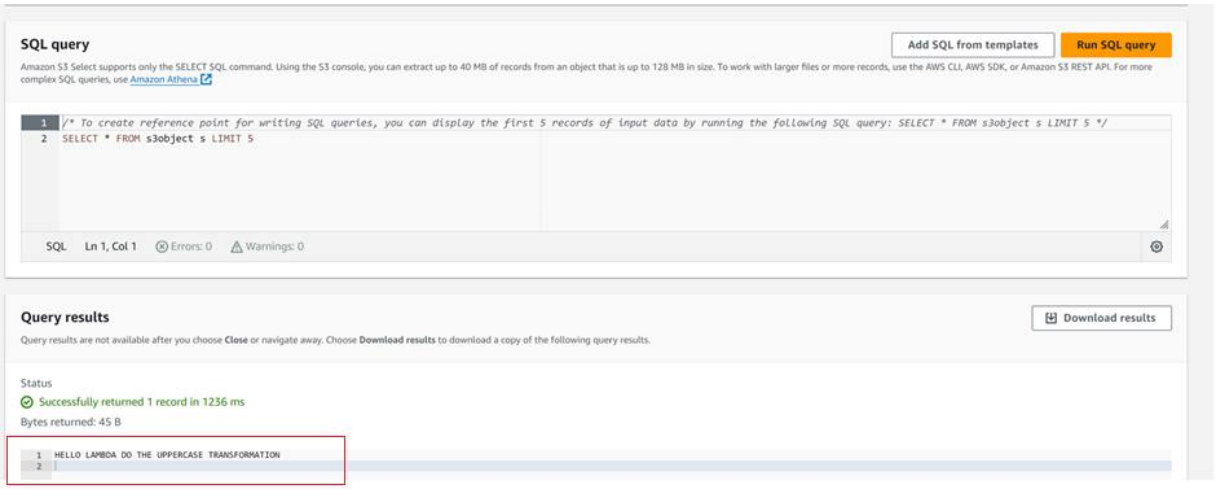


Now, my lambda function will read the file from the input folder and right the transformed file in the output folder.



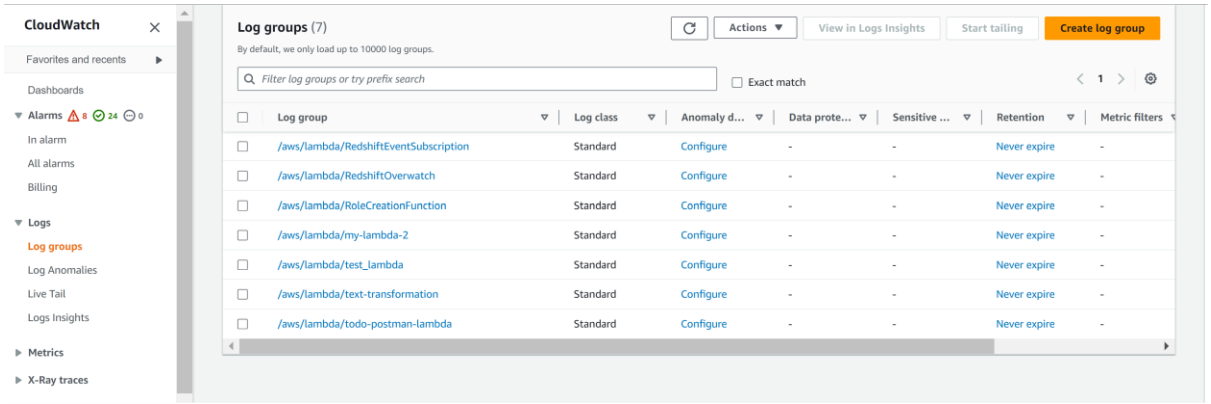






As you can see, our text has been transformed into uppercase.

We can check CloudWatch for the logs.



CloudWatch

Favorites and recents

Dashboards

Alarms 8 24 0

In alarm

All alarms

Billing

Logs

Log groups

Log Anomalies

Live Tail

Logs Insights

Metrics

X-Ray traces

Events

Application Signals

Network monitoring

Insights

Container Insights

2 days ago

Retention
Never expire

Stored bytes
13.18 KB

KMS key ID
-

Log streams | Tags | Anomaly detection | Metric filters | Subscription filters | Contributor Insights | Data protection

Log streams (7)

Delete | Create log stream | Search all log streams

Filter log streams or try prefix search

Exact match | Show expired | Info

1

Log stream	Last event time
2024/02/22/[$LATEST]1396aa33ff22401687c29fd450ed1289	2024-02-22 17:45:12 (UTC+05:45)
2024/02/20/[$LATEST]984935186 34c4d998c888545124eba64	2024-02-20 16:39:45 (UTC+05:45)
2024/02/20/[$LATEST]30b7d816cc36480f814c52933ab1725b	2024-02-20 16:35:33 (UTC+05:45)
2024/02/20/[$LATEST]f2cffbc76e354d2f8595570e10d0d976	2024-02-20 16:32:11 (UTC+05:45)
2024/02/20/[$LATEST]c911feb0c2a945559414d7c0e873787b	2024-02-20 16:28:02 (UTC+05:45)
2024/02/20/[$LATEST]f8b7b2602a17414cba99a28e9442bddb	2024-02-20 16:08:21 (UTC+05:45)
2024/02/20/[$LATEST]bcd8178a054742b19f24bf3478f58571	2024-02-20 16:00:45 (UTC+05:45)

CloudWatch

Favorites and recents

Dashboards

Alarms 8 24 0

In alarm

All alarms

Billing

Logs

Log groups

Log Anomalies

Live Tail

Logs Insights

Metrics

CloudWatch > Log groups > /aws/lambda/text-transformation > 2024/02/22/[$LATEST]1396aa33ff22401687c29fd450ed1289

Log events

Actions | Start tailing | Create metric filter

You can use the filter bar below to search for and match terms, phrases, or values in your log events. Learn more about filter patterns

Filter events

Clear | 1m | 30m | 1h | 12h | Custom | Local timezone | Display

Timestamp	Message
	No older events at this moment. *Retry*
2024-02-22T17:45:12.173+05:45	INIT_START Runtime Version: python:3.12.v19 Runtime Version ARN: arn:aws:lambda:us-east-1::runtime:a79ae1de439e89e7a1dc89465a221e8fe9bb3c495c3ff8e61b…
2024-02-22T17:45:12.785+05:45	START RequestId: 5f11f5fc-18fc-4369-8b8a-47857df69d82 Version: $LATEST
2024-02-22T17:45:13.467+05:45	END RequestId: 5f11f5fc-18fc-4369-8b8a-47857df69d82
2024-02-22T17:45:13.467+05:45	REPORT RequestId: 5f11f5fc-18fc-4369-8b8a-47857df69d82 Duration: 681.48 ms Billed Duration: 682 ms Memory Size: 128 MB Max Memory Used: 81 MB Init Du…
	No newer events at this moment. *Auto retry paused. Resume*