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.

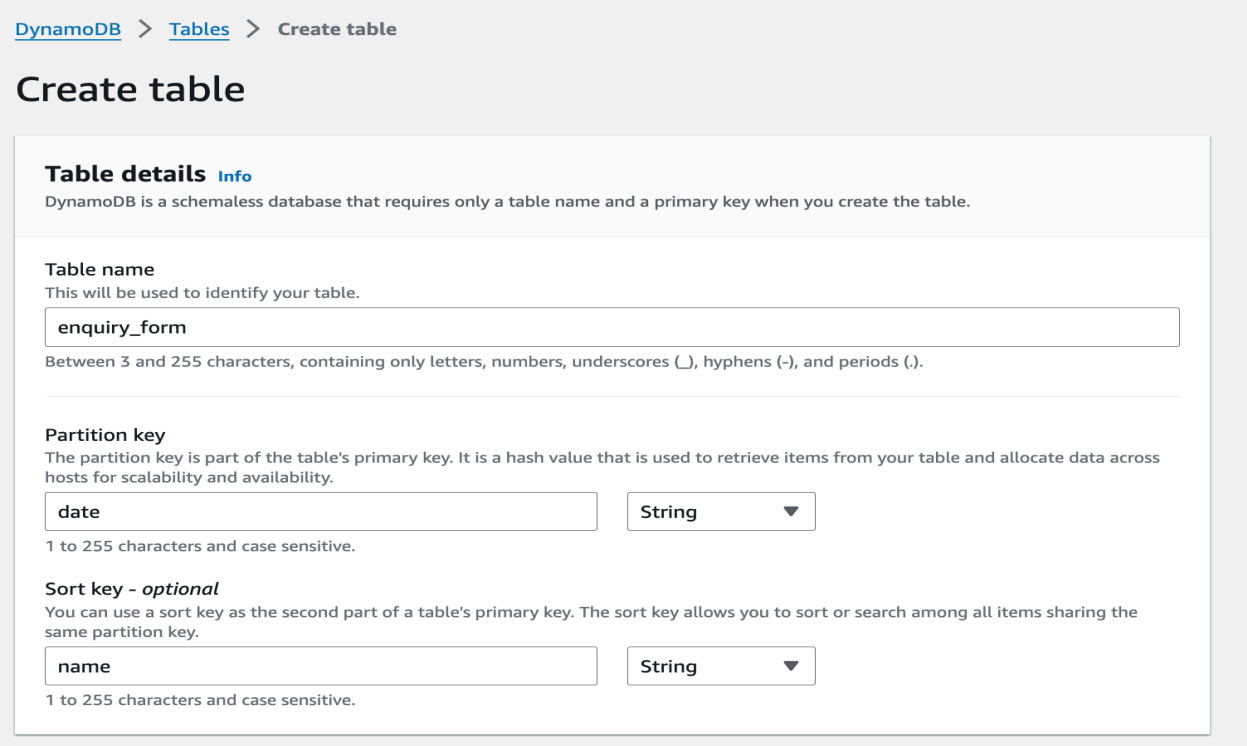
API Gateway: Set up API Gateway to create RESTful endpoints that trigger the Lambda functions.

Frontend Hosting: Host a static website on S3 that interacts with the backend via API Gateway.

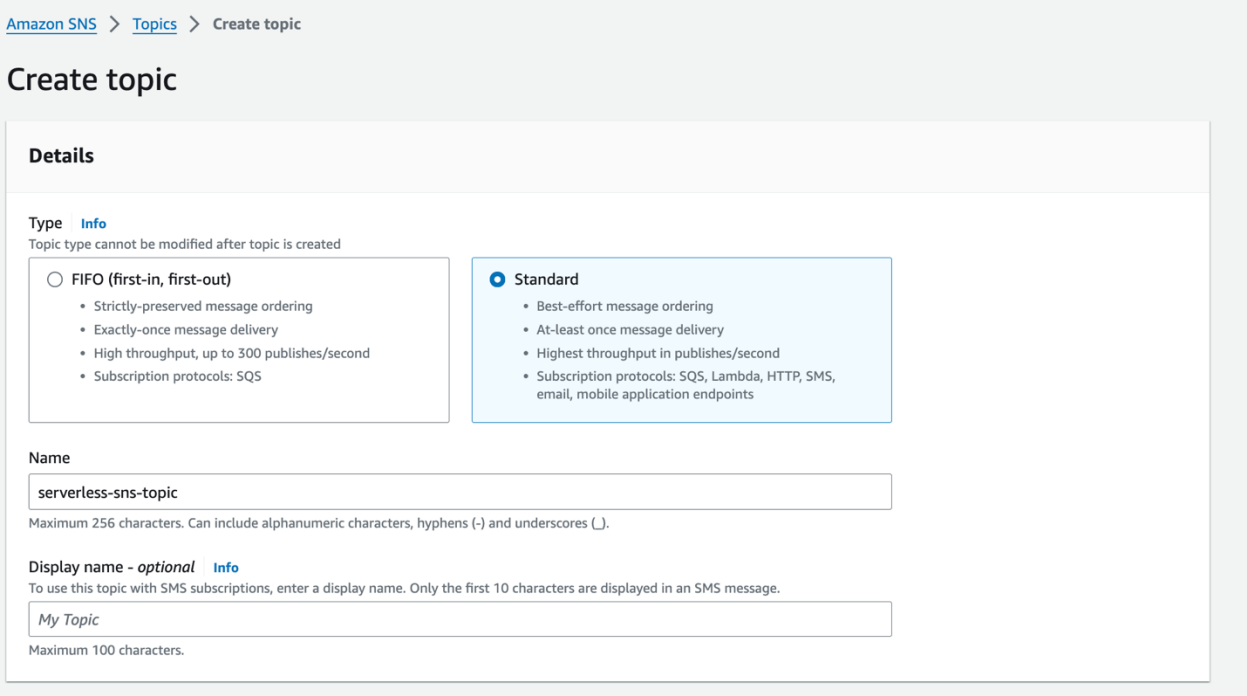
Integration: Ensure that the frontend can successfully send requests to the backend and display responses.

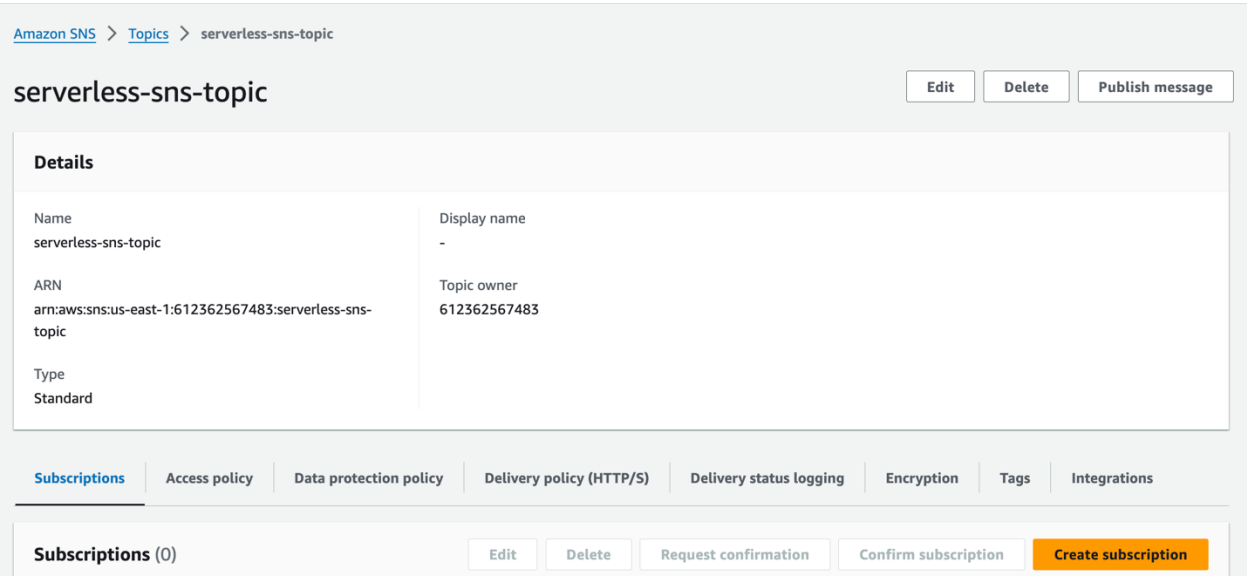
Goal: Understand the basics of building and connecting serverless backend services with a static frontend, enabling a fully serverless web application.\

1.1. Create a table in DynamoDb



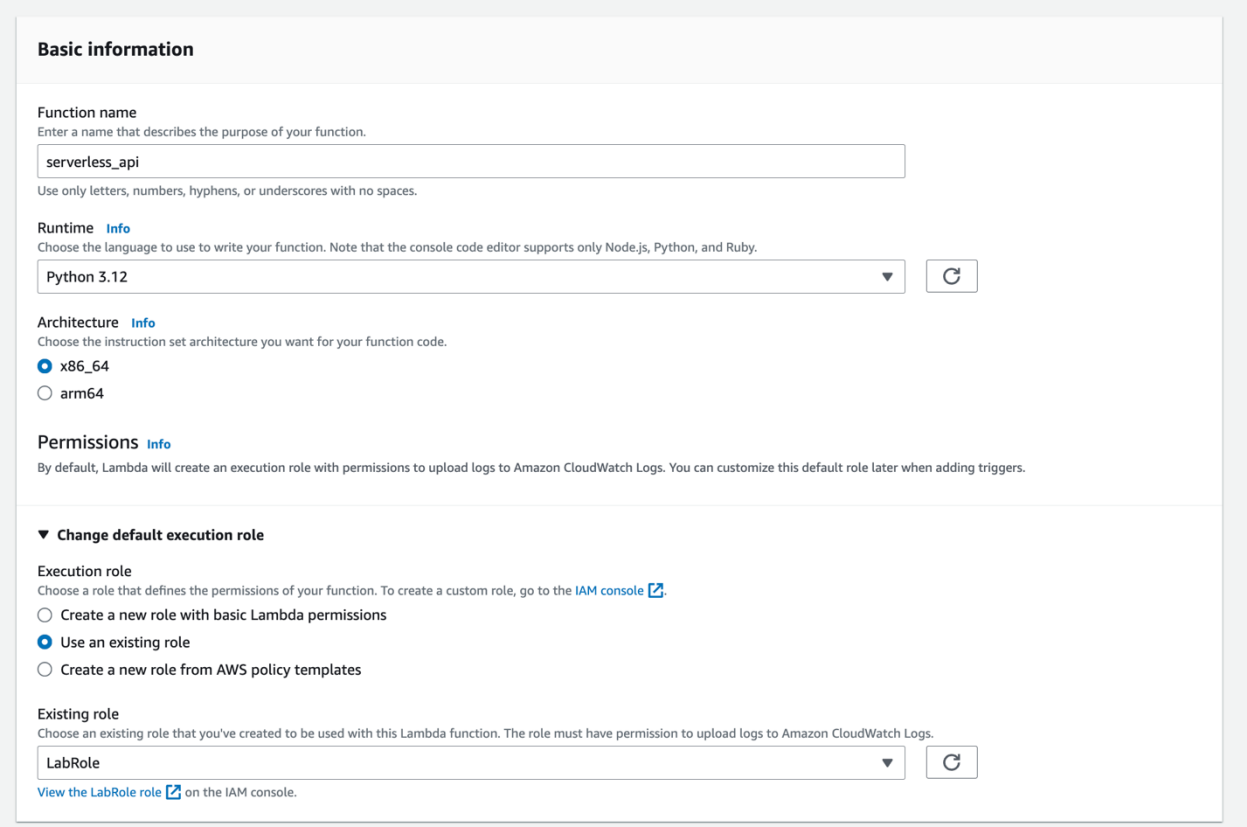
1.2. SNS Topic


Amazon SNS > Topics > Create topic
Create topic
Details
Type Info
Topic type cannot be modified after topic is created
FIFO (first-in, first-out)
• Strictly-preserved message ordering
• Exactly-once message delivery
• High throughput, up to 300 publishes/second
• Subscription protocols: SQS
Standard
• Best-effort message ordering
• At-least once message delivery
• Highest throughput in publishes/second
• Subscription protocols: SQS, Lambda, HTTP, SMS, email, mobile application endpoints
Name
serverless-sns-topic
Maximum 256 characters. Can include alphanumeric characters, hyphens (-) and underscores (_).
Display name - optional Info
To use this topic with SMS subscriptions, enter a display name. Only the first 10 characters are displayed in an SMS message.
My Topic
Maximum 100 characters.


Amazon SNS > Topics > serverless-sns-topic
serverless-sns-topic
Edit
Delete
Publish message
Details
Name
serverless-sns-topic
Display name
-
ARN
arn:aws:sns:us-east-1:612362567483:serverless-sns-topic
Topic owner
612362567483
Type
Standard
Subscriptions
Access policy
Data protection policy
Delivery policy (HTTP/S)
Delivery status logging
Encryption
Tags
Integrations
Subscriptions (0)
Edit
Delete
Request confirmation
Confirm subscription
Create subscription

1.3. Create lambda

Lambda function is created using existing user role.

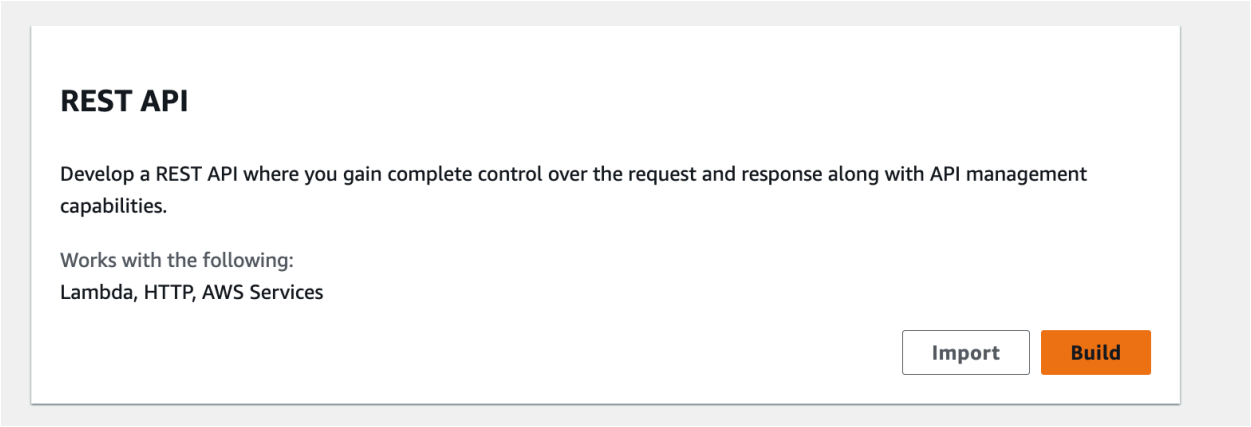


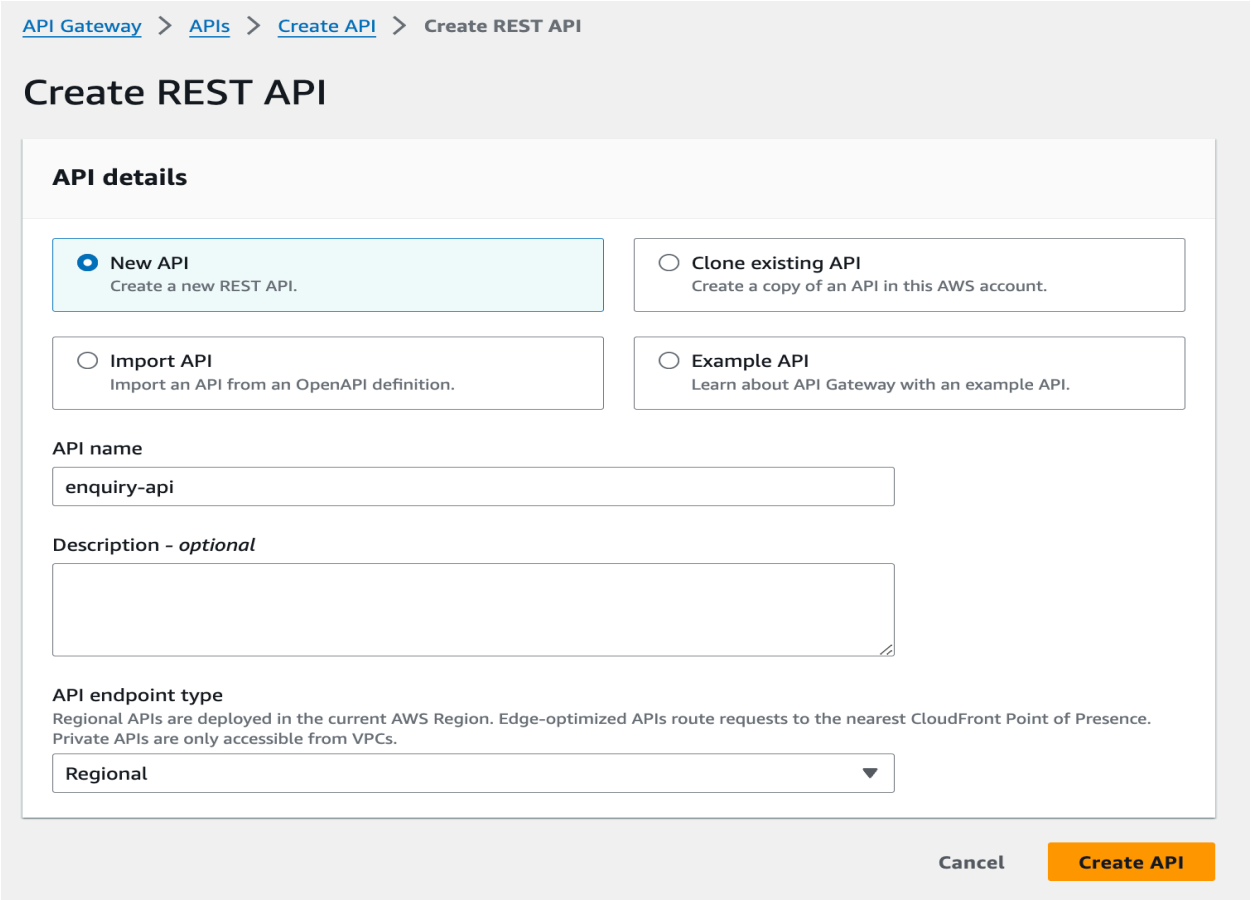
After that, post request implementation is done

```python
import json
import boto3
import datetime

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('enquiry_form')

def sns_notification(sns_arn, sns_message):
    client = boto3.client('sns')
    response = client.publish(TopicArn=sns_arn, Message=sns_message)
    print('response:', response)
    return response


def lambda_handler(event, context):
    try:
        print('Received event:', json.dumps(event))
        current_date = datetime.datetime.now().strftime("%Y-%m-%d")
        name = event['name']
        email = event['email']
        message = event['message']
        item ={
            'date':current_date,
            'name':name,
            'email':email,
            'message':message
        }

        table.put_item(Item=item)
        response = {
            "statusCode": 200,
            "data": json.dumps({"message": "Data is added successfully"})
        }
        #sns implementation
        # arn = "arn:aws:sns:us-east-1:612362567483:serverless-sns-topic"
        # message=f"New data stored"
        # resource = sns_notification(arn,message)
    except Exception as e:
        print("Error:", e)
        response = {
            "statusCode": 500,
            "data": json.dumps({"message": "Internal Server Error"})
        }
    return response

```

1.4. Create API Gateway

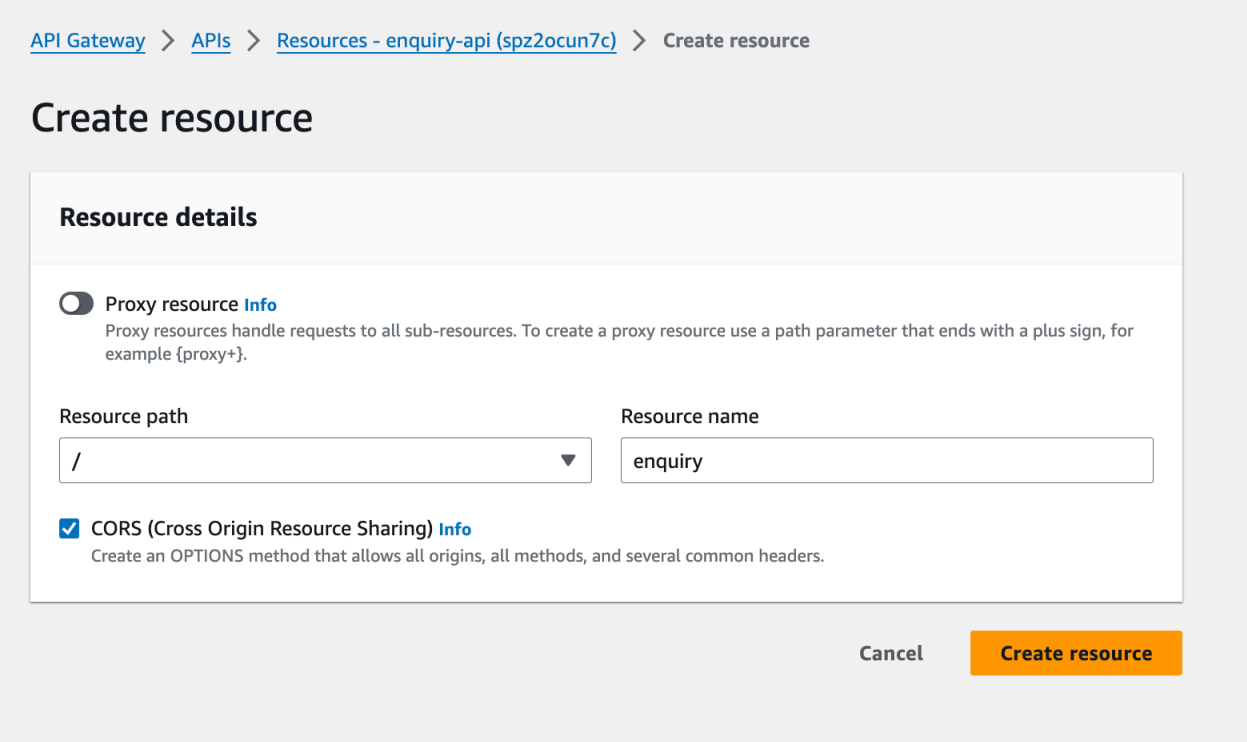
Reuqired REST API configuration is added





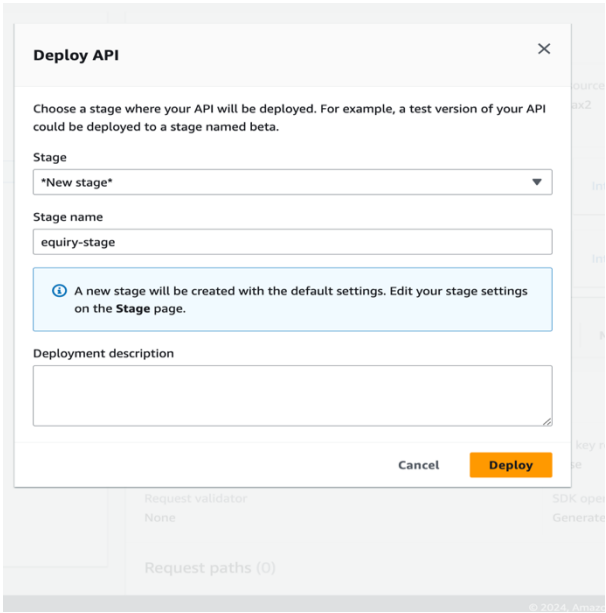
Steps implemented in API gateways are mentioned below:

i. Creating Resource

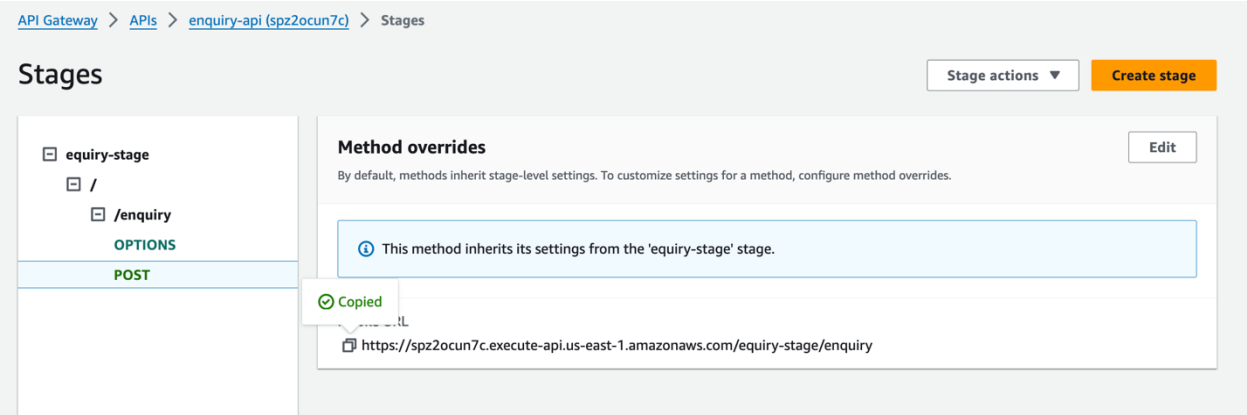


ii. Crete a POST Method and add recently created lambda function in the configuration

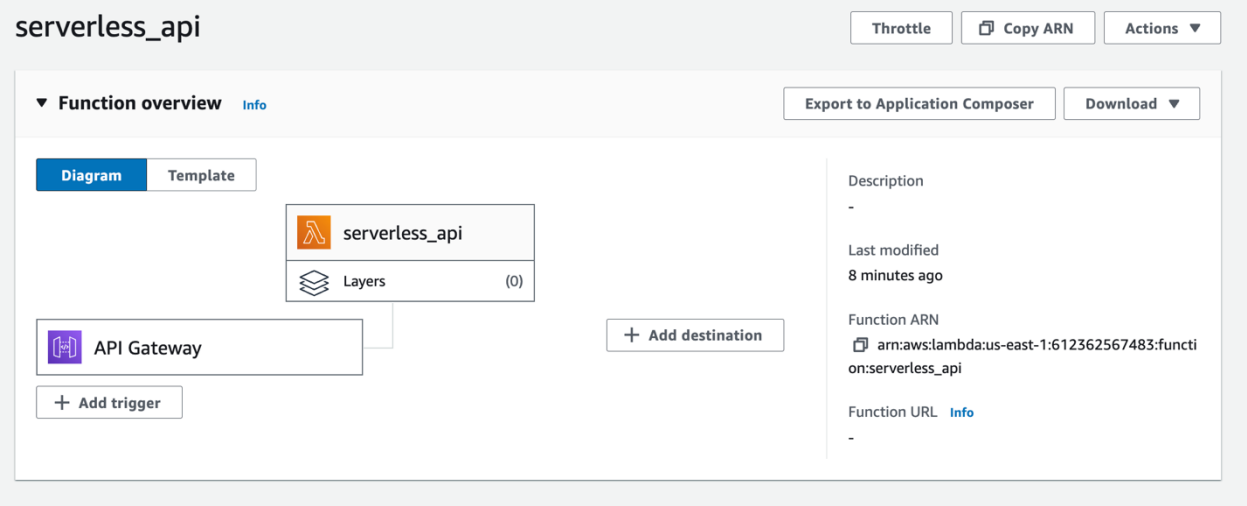
iii. Deploy the API. For this purpose a new stage is created and then deployed.



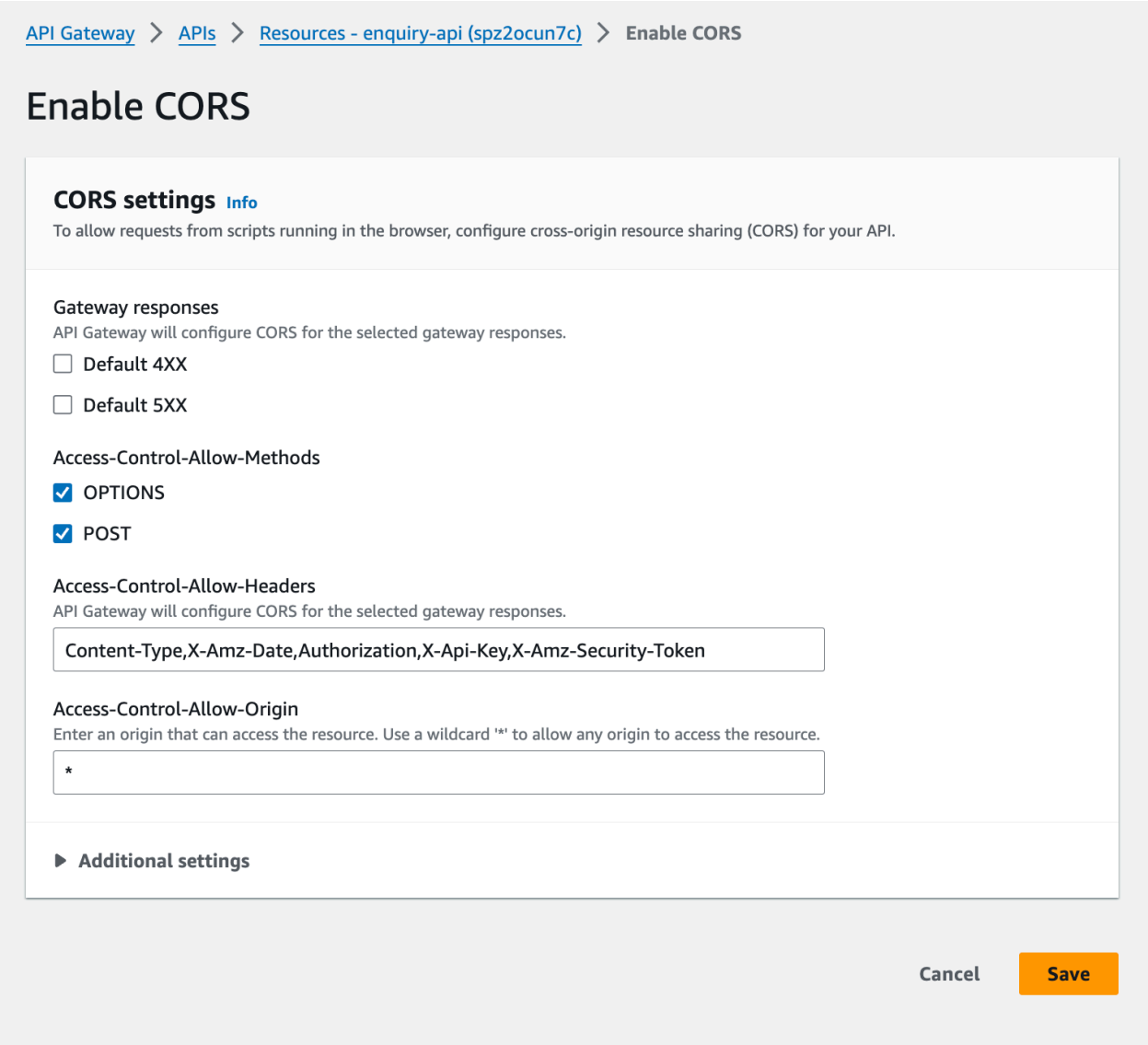
iv. Invoke URL of the post method is copied from the created stage



v. Now in lambda funtion, we can observe that API Gateway is added.

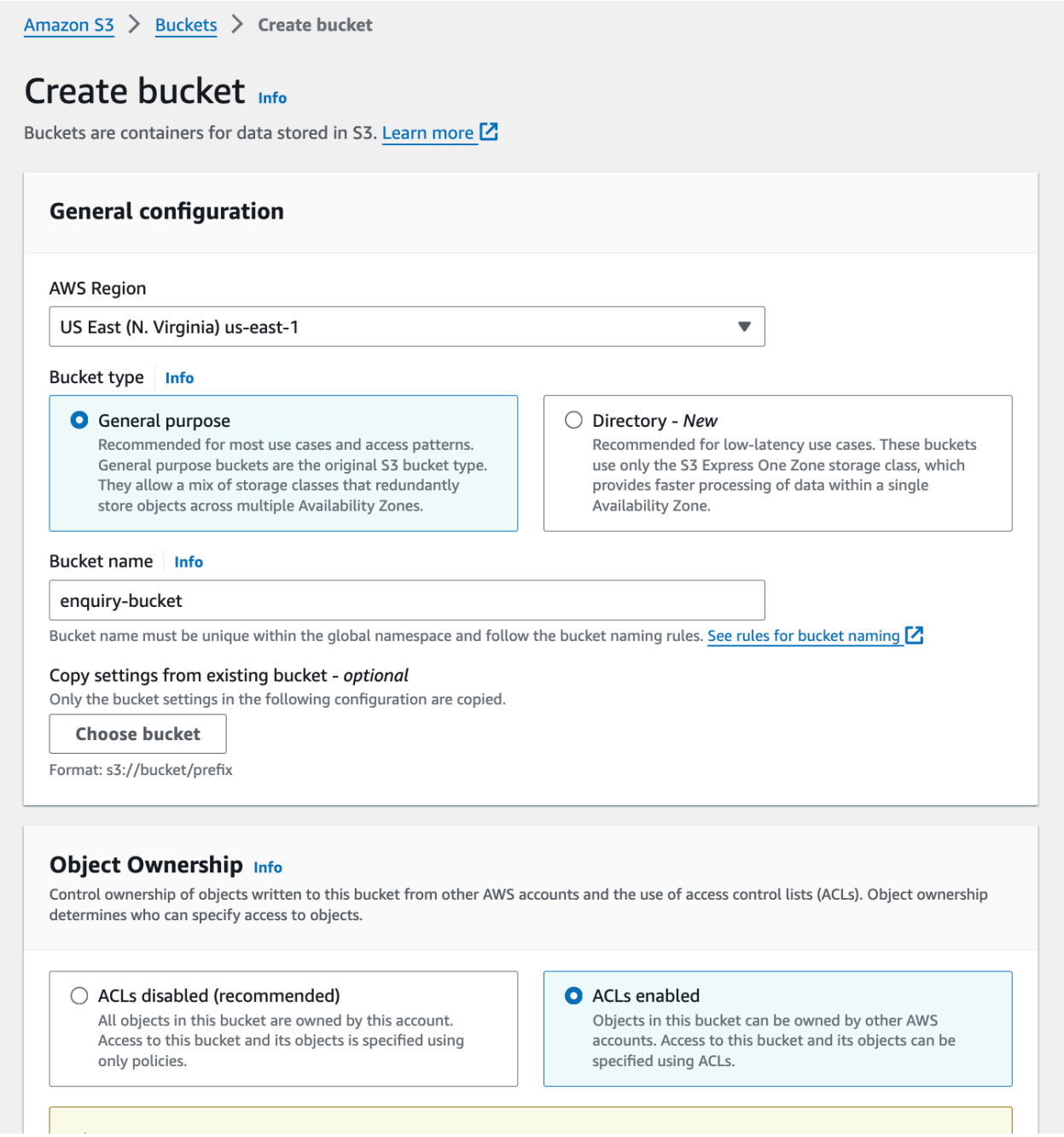


vi. The API CORS is enabled.

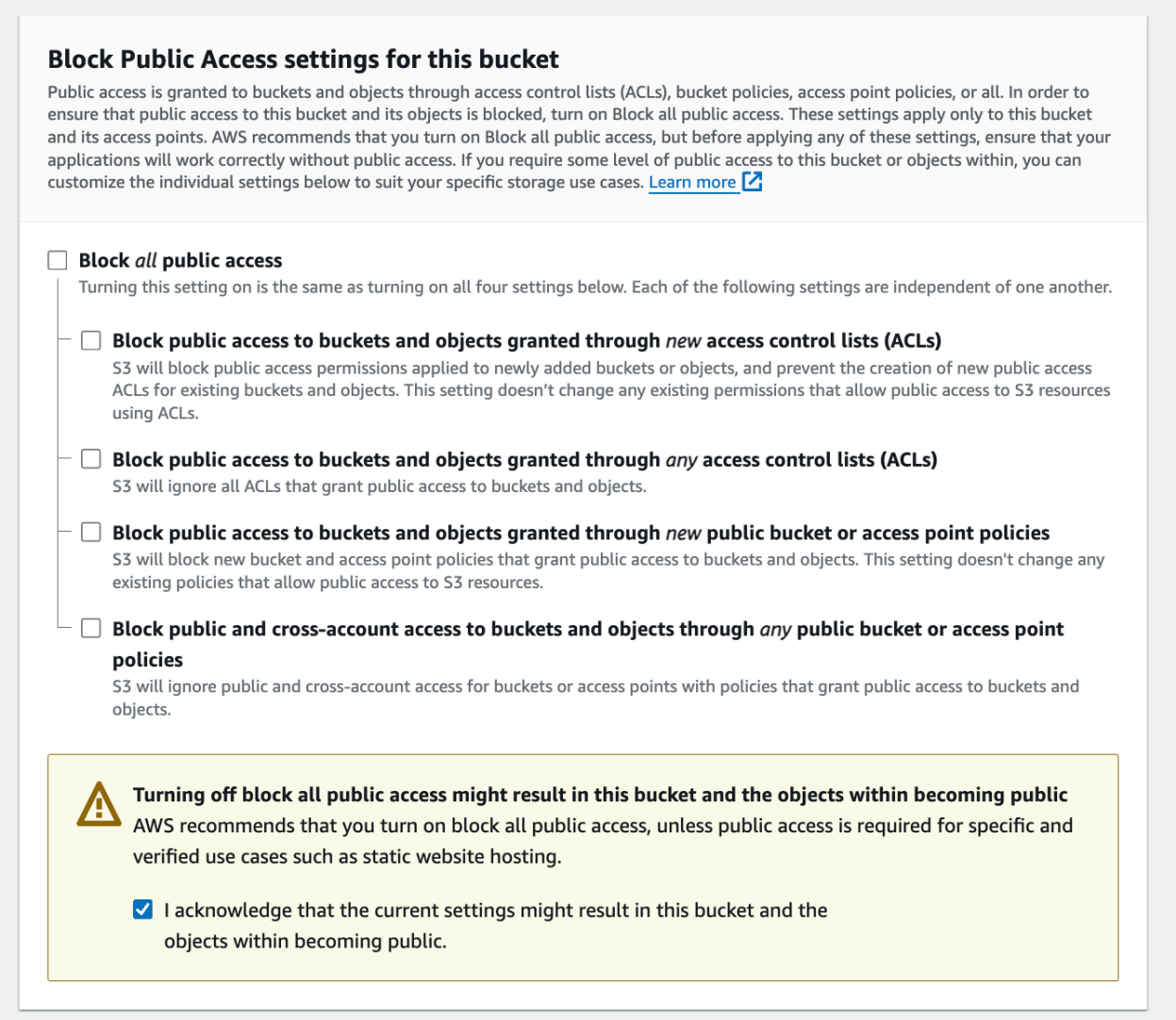


1.5. S3 bucket creation and uploading the html file

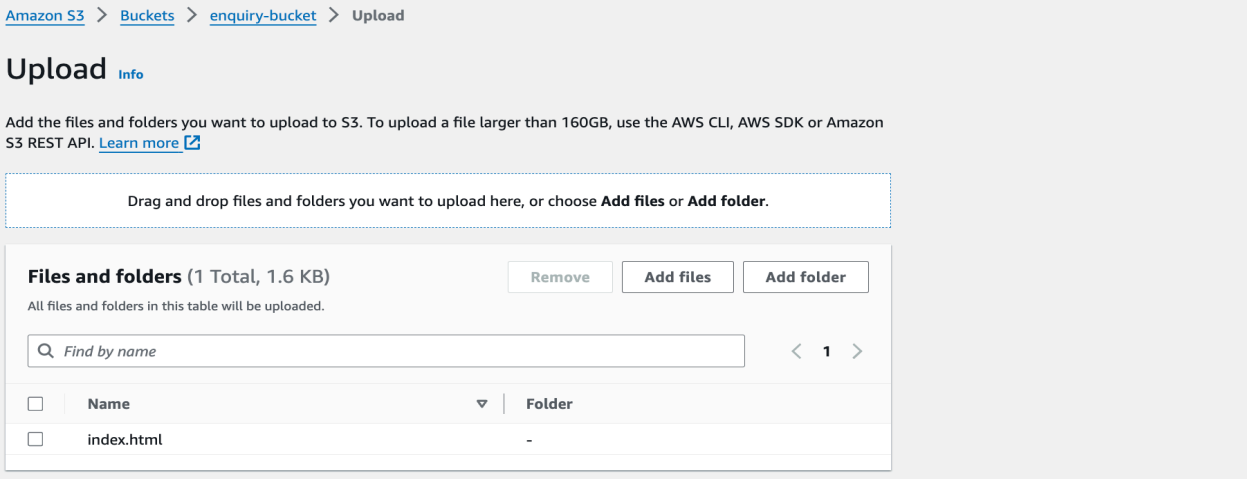
Adding all the required configuration is S3 bucket. “ACLs enabled” is choosen in object ownership so that bucket is publicy available with the url.

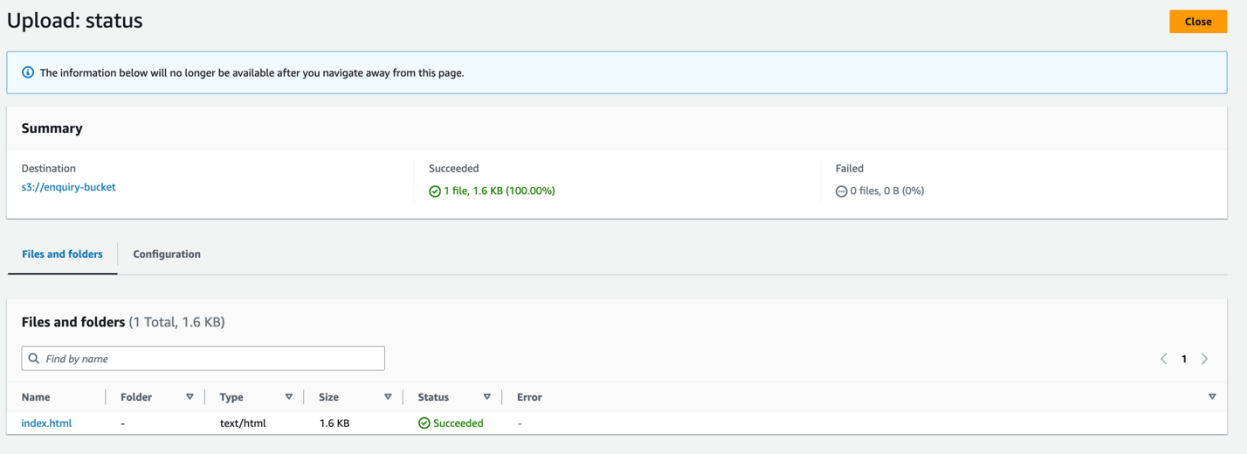


Block all public acess is removed for getting access through webiste.

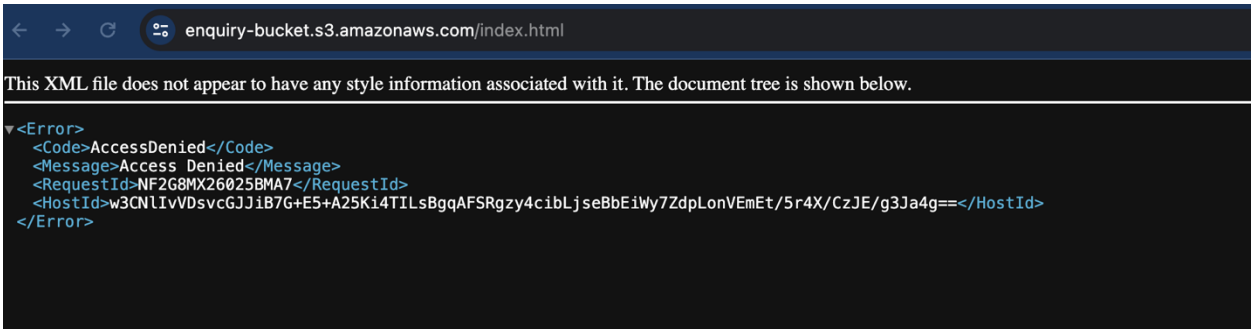


After creation of S3 bucket, html file containing enquiry form is uploaded.

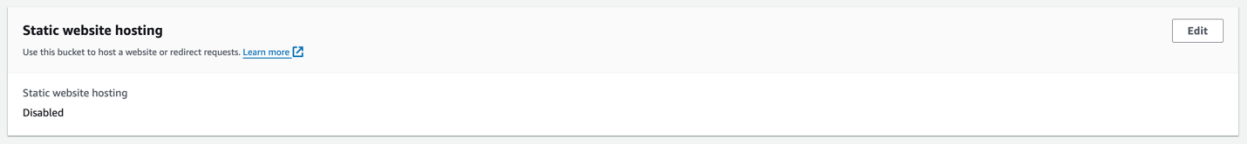


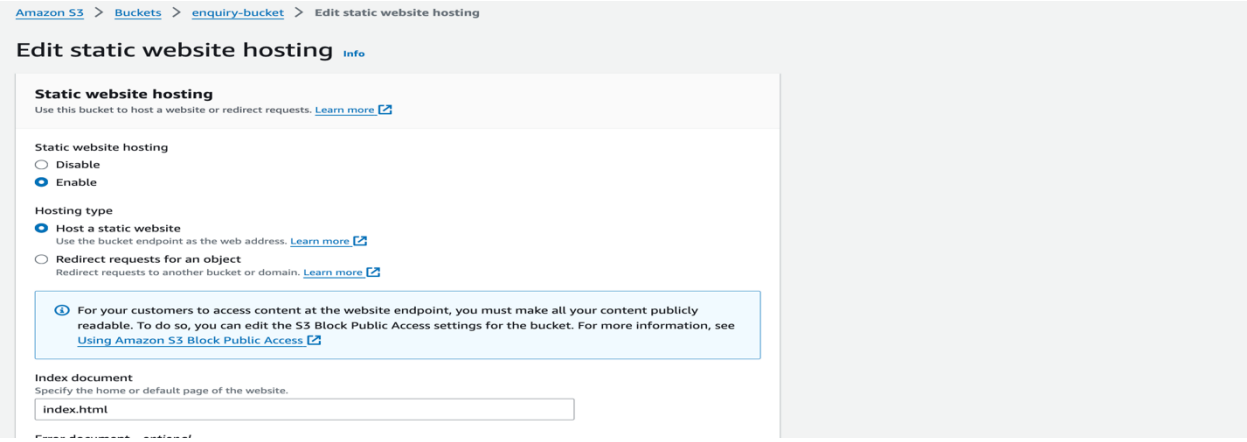


When the url of uploaded file is browsed in web, access denied message is shown.

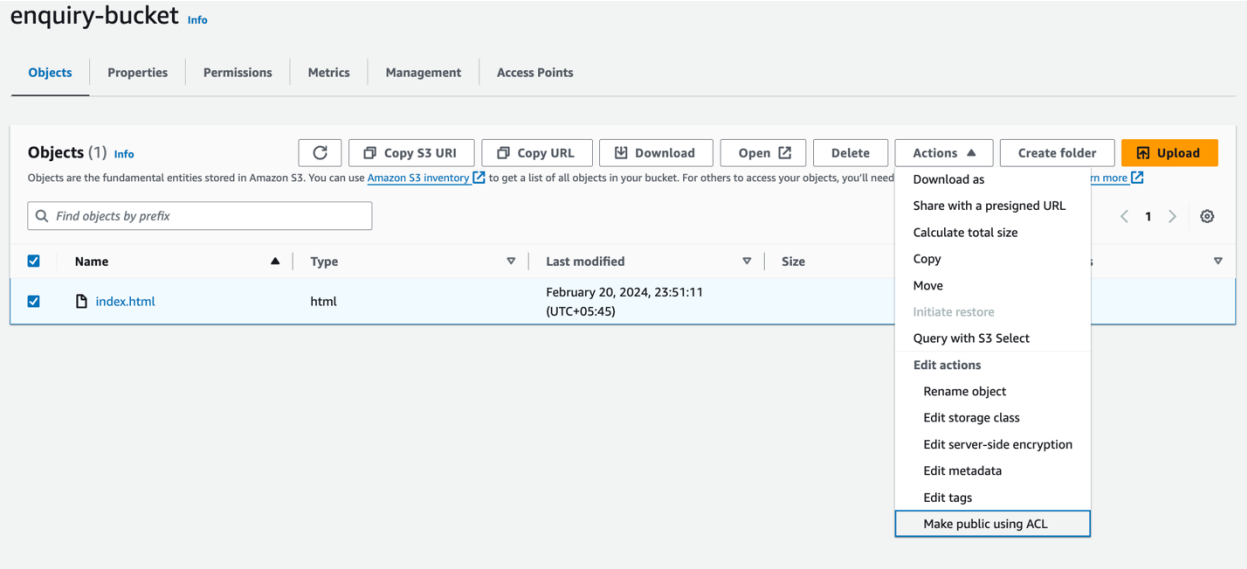


So, now the website is hosted from properties tab of created S3 bucket. Initially the static website hosting is disabled and for hosting it's status is updated to "Enable".

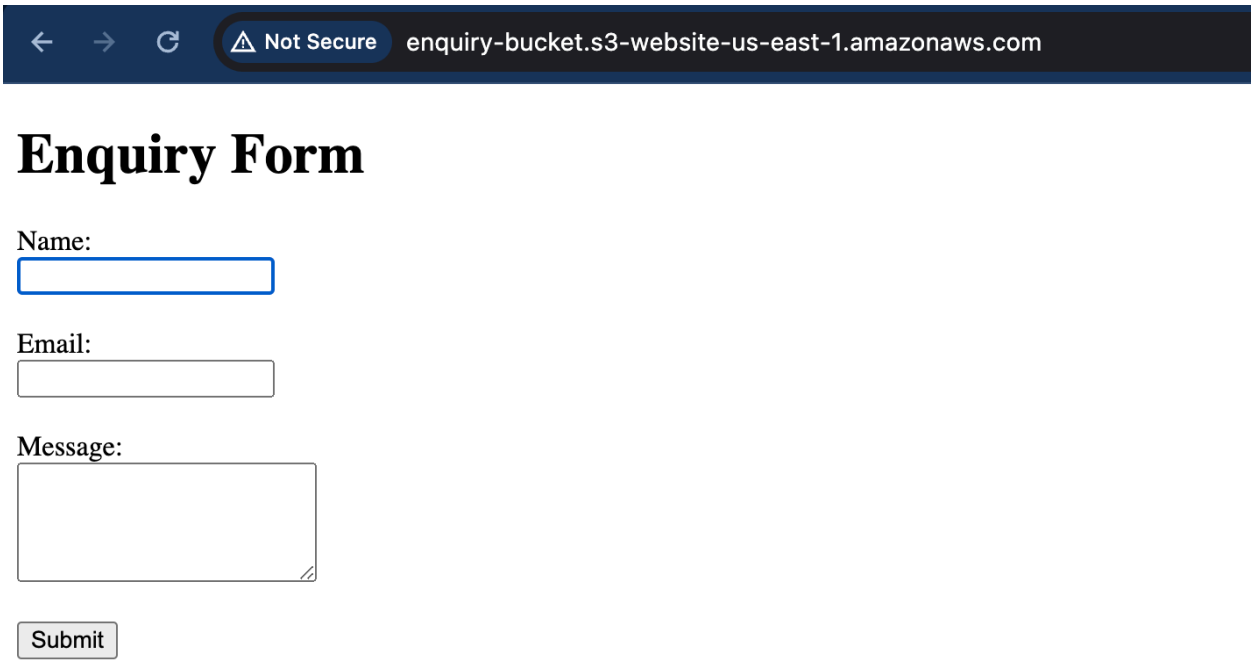




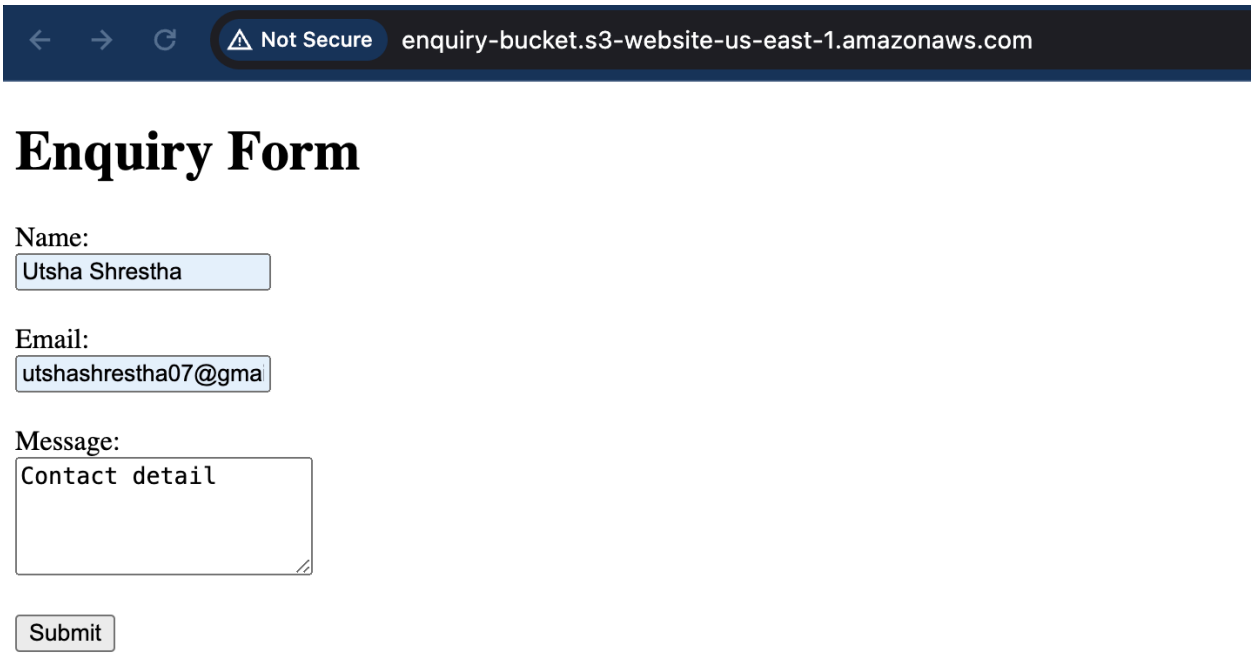
After that, the uploded file should give permission to make public using ACL.

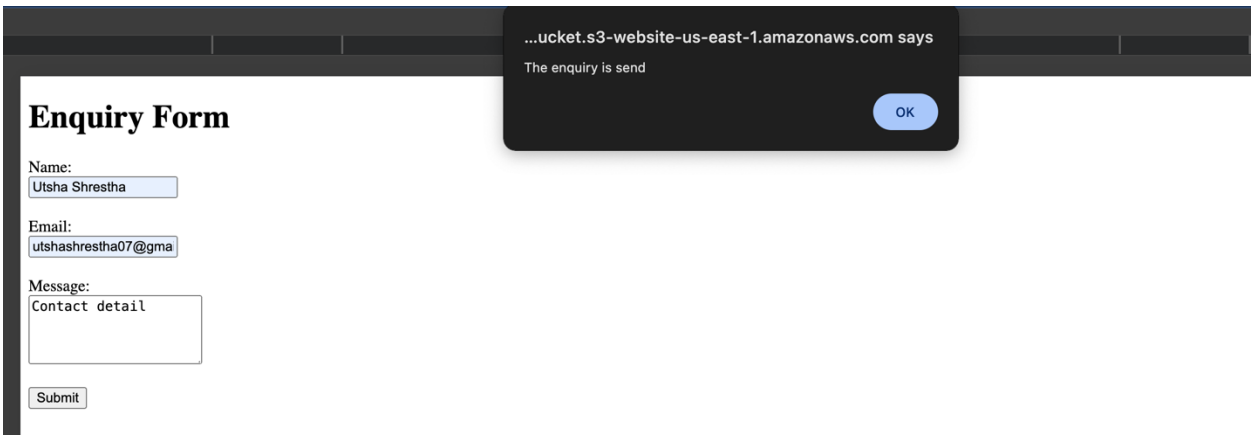


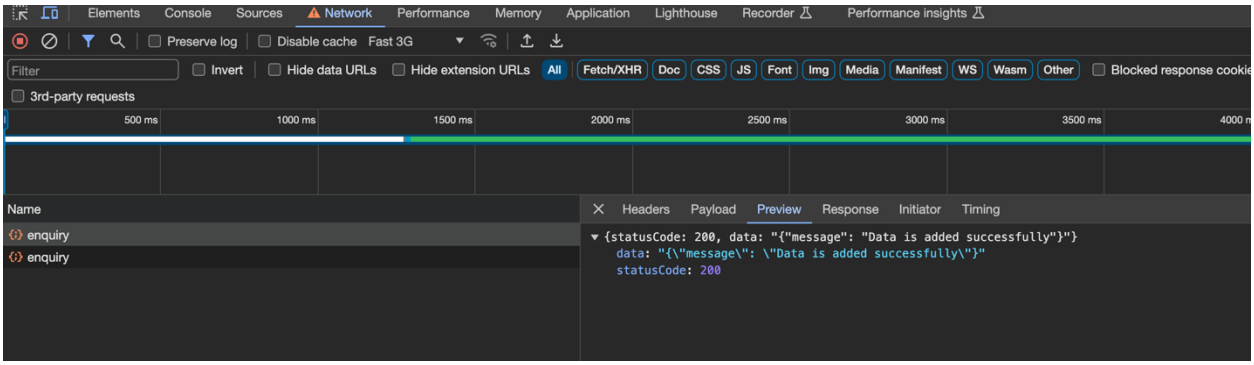
Now, the hosted website can be accessed.

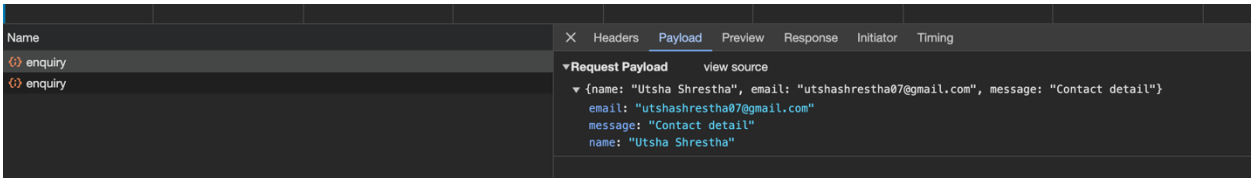


The testing of implemented task is checked. Below we can observe the user input and the response of the integrated API. I have checked by inspecting the request in browser.

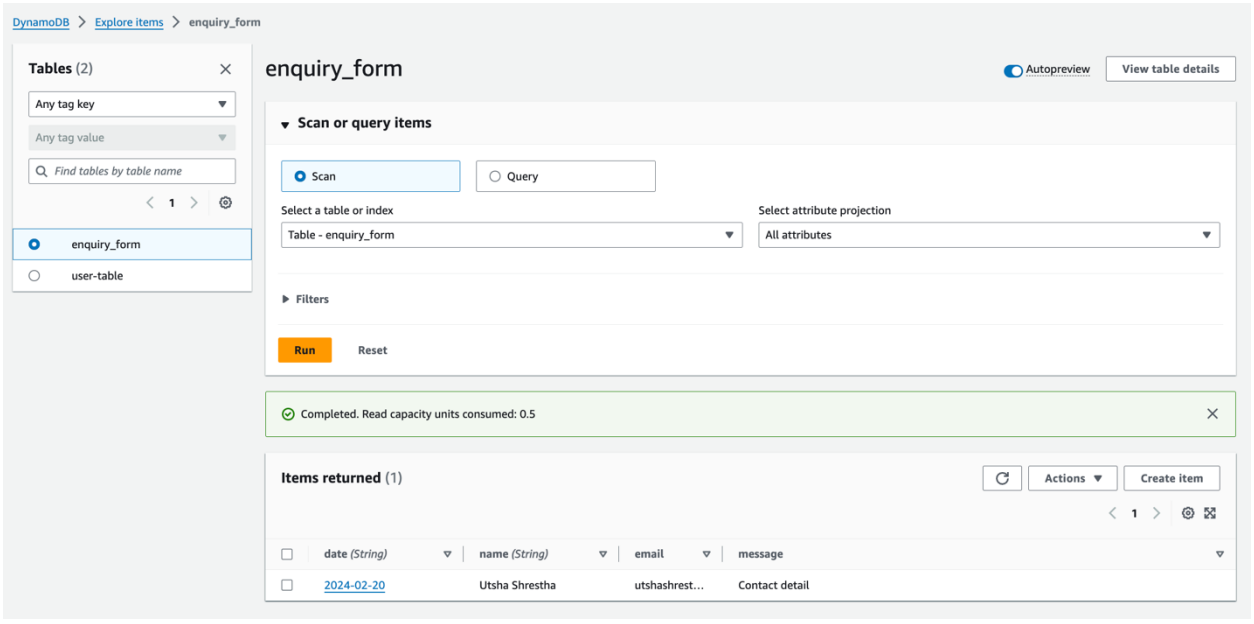




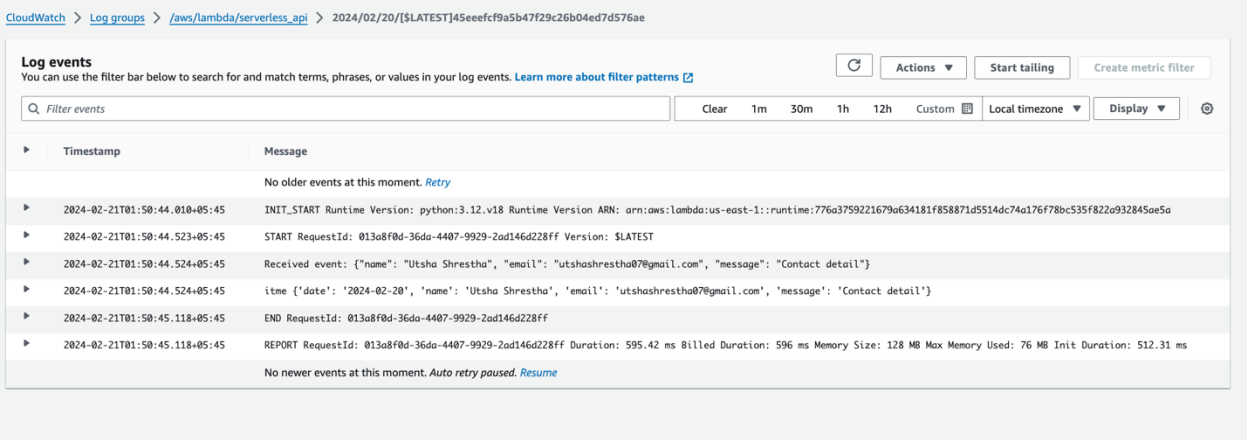




The items in DynamoDB is inserted too.



The logs of implemented task can be observed from cloud watch logs of lambda function.



2. Creating a Serverless API

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

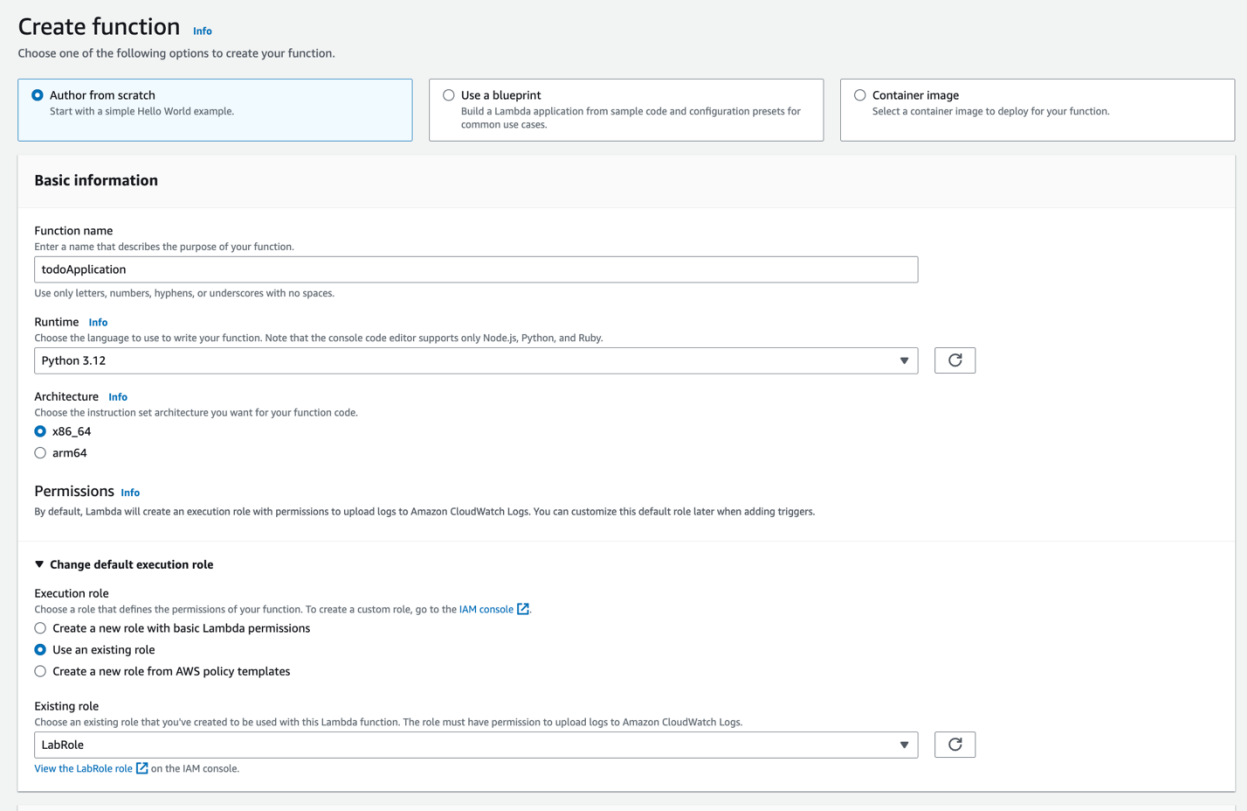
Approach:

- **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.

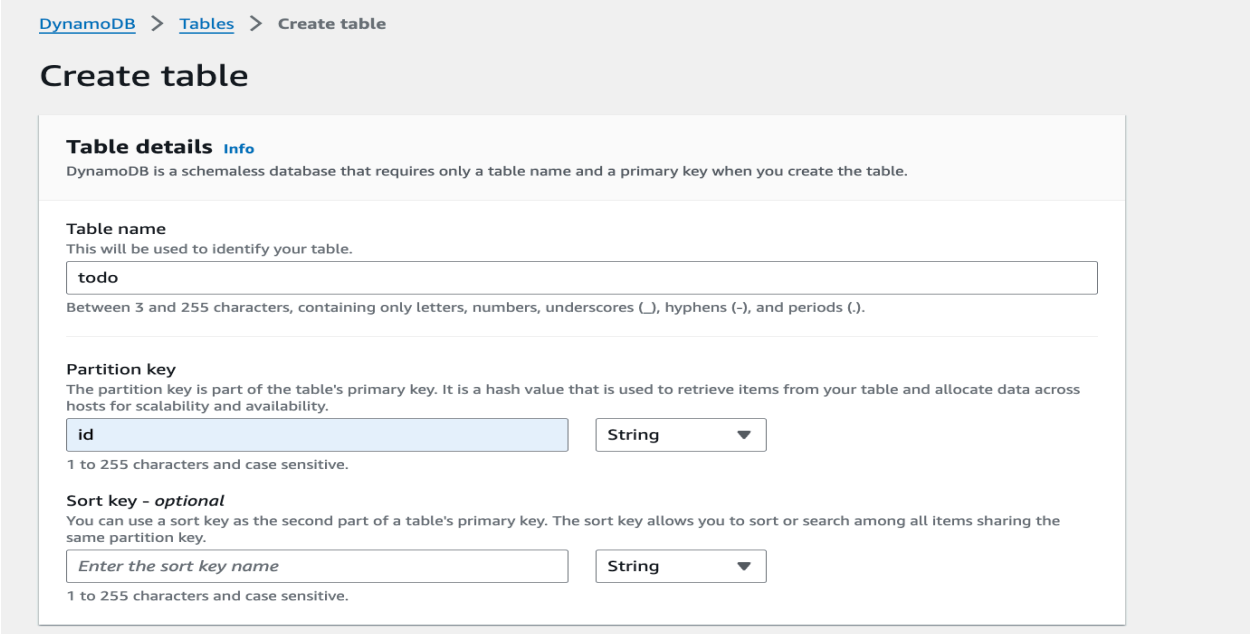
Goal: Gain hands-on experience in building and deploying a serverless API, understanding the integration between Lambda and API Gateway.

2.1. Create lambda function

Lambda functions with required configuration is created

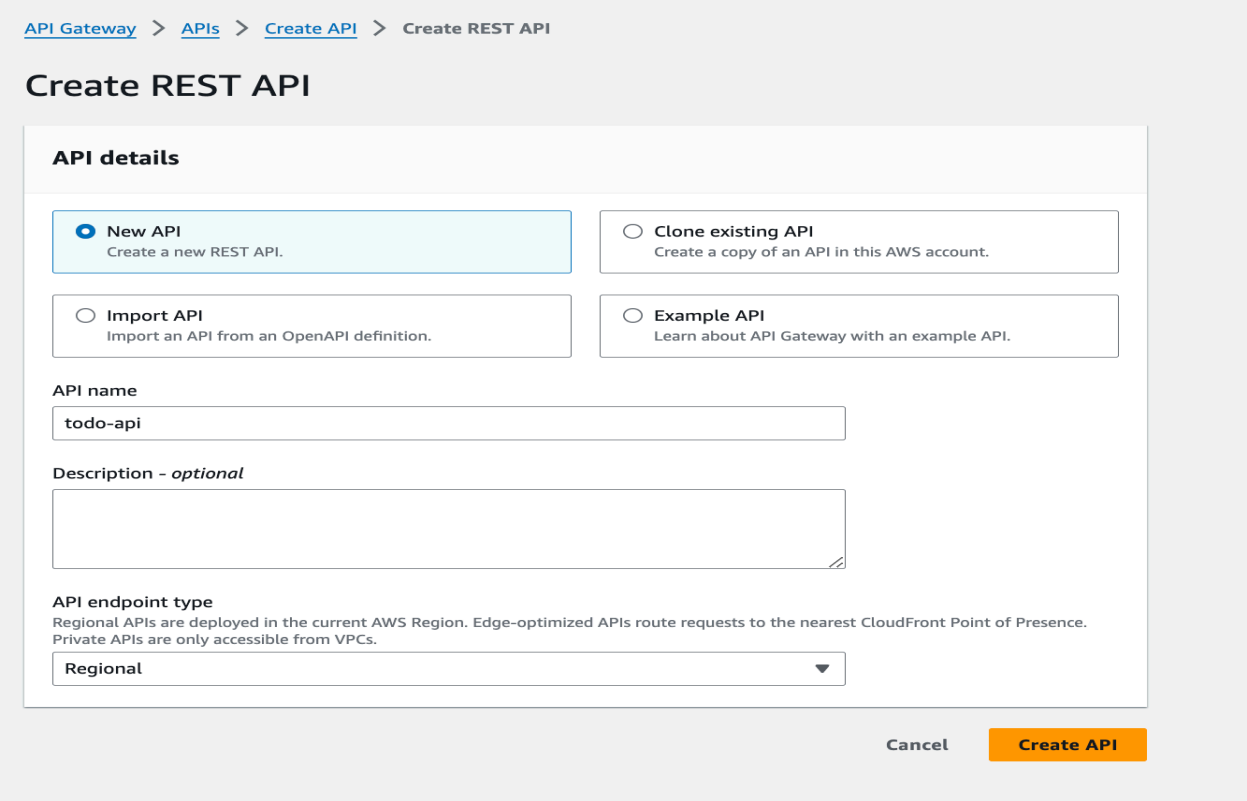


2.2. Create a table in DynamoDB

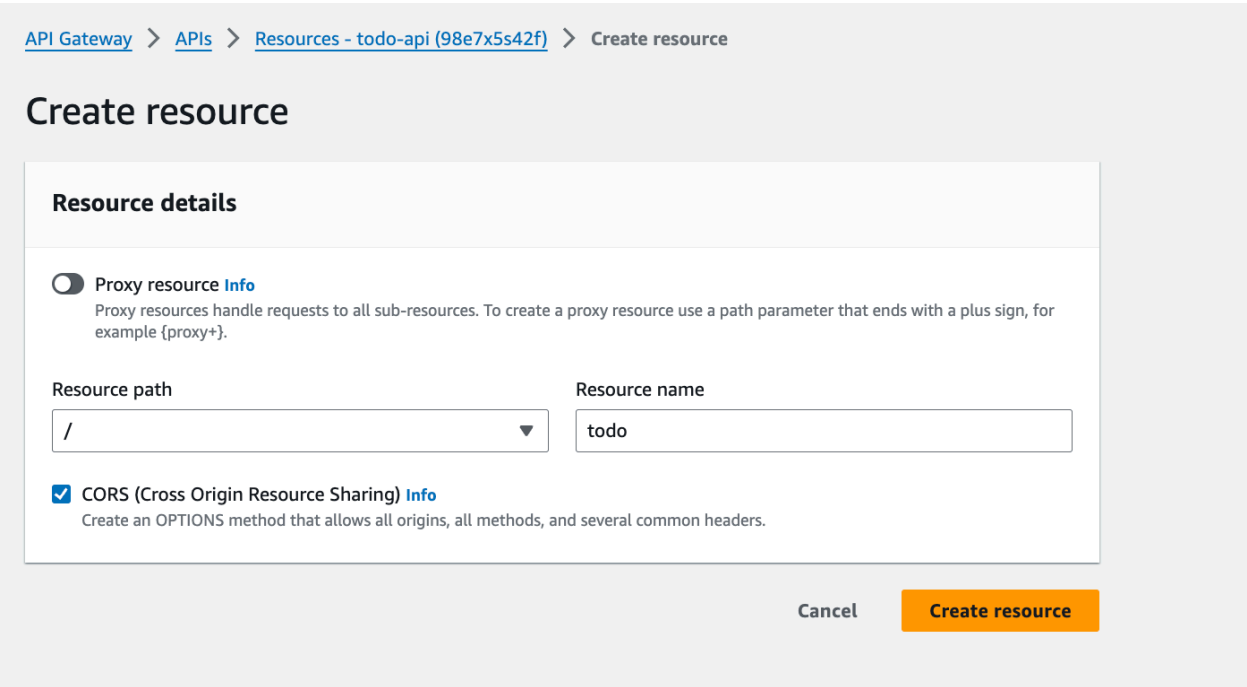


2.3. API Gateway configuration

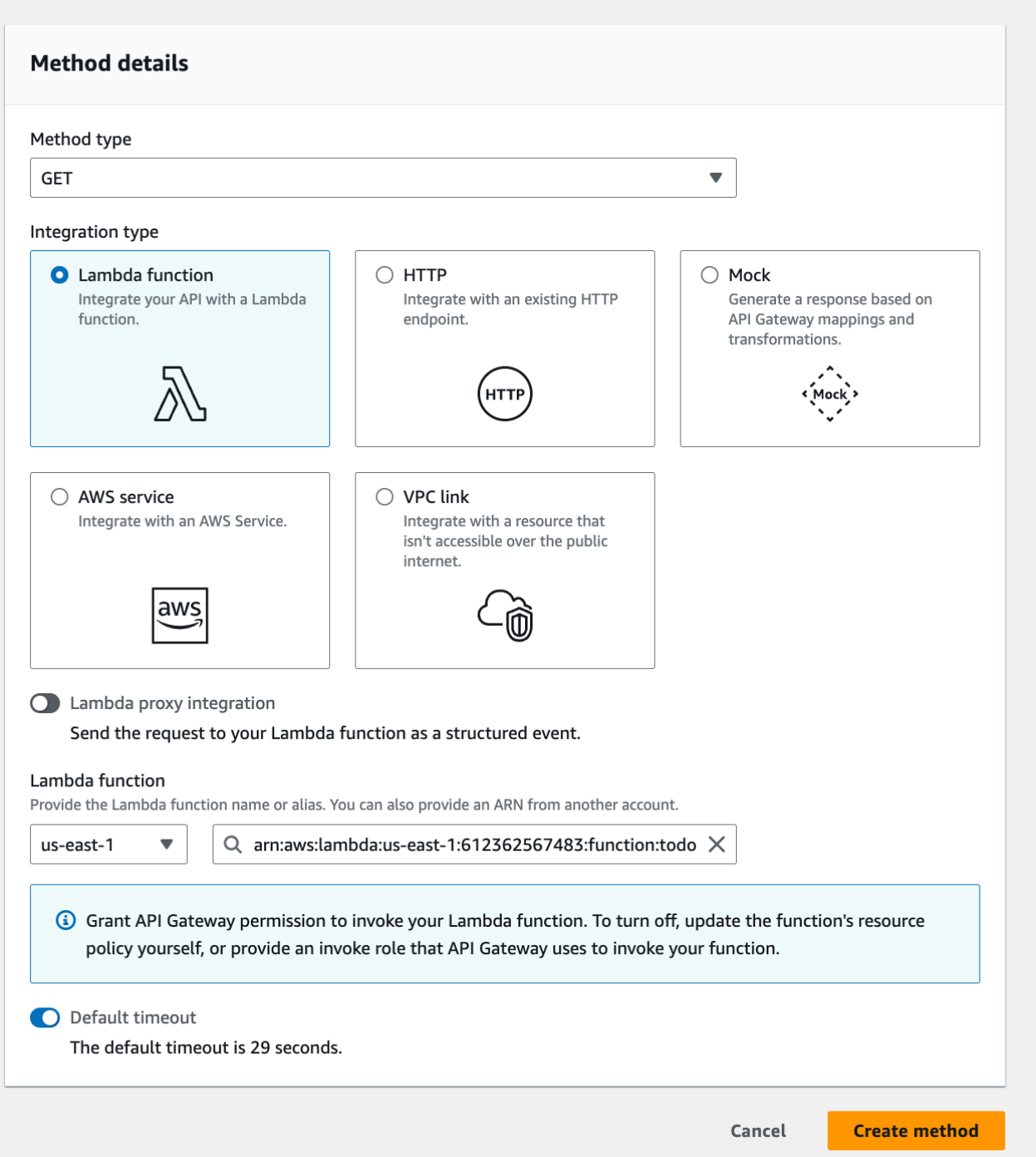
Create a rest api



Creating a resource for the API



Creating GET, POST, PUT, DELETE methods of API



Create method

Method details

Method type

POST

Integration type

- **Lambda function**
  Integrate your API with a Lambda function.
  
- **HTTP**
  Integrate with an existing HTTP endpoint.
  

- **Mock**
  Generate a response based on API Gateway mappings and transformations.
  

- **AWS service**
  Integrate with an AWS Service.
  

- **VPC link**
  Integrate with a resource that isn't accessible over the public internet.
  

Lambda proxy integration

Send the request to your Lambda function as a structured event.

Lambda function

Provide the Lambda function name or alias. You can also provide an ARN from another account.

us-east-1

arn:aws:lambda:us-east-1:612362567483:function:todo

Grant API Gateway permission to invoke your Lambda function. To turn off, update the function's resource policy yourself, or provide an invoke role that API Gateway uses to invoke your function.

Default timeout

The default timeout is 29 seconds.

Cancel

Create method

Create method

Method details

Method type

PUT

Integration type

- **Lambda function**
  Integrate your API with a Lambda function.
  
- **HTTP**
  Integrate with an existing HTTP endpoint.
  

- **Mock**
  Generate a response based on API Gateway mappings and transformations.
  

- **AWS service**
  Integrate with an AWS Service.
  

- **VPC link**
  Integrate with a resource that isn't accessible over the public internet.
  

Lambda proxy integration

Send the request to your Lambda function as a structured event.

Lambda function

Provide the Lambda function name or alias. You can also provide an ARN from another account.

us-east-1

arn:aws:lambda:us-east-1:612362567483:function:todo

Grant API Gateway permission to invoke your Lambda function. To turn off, update the function's resource policy yourself, or provide an invoke role that API Gateway uses to invoke your function.

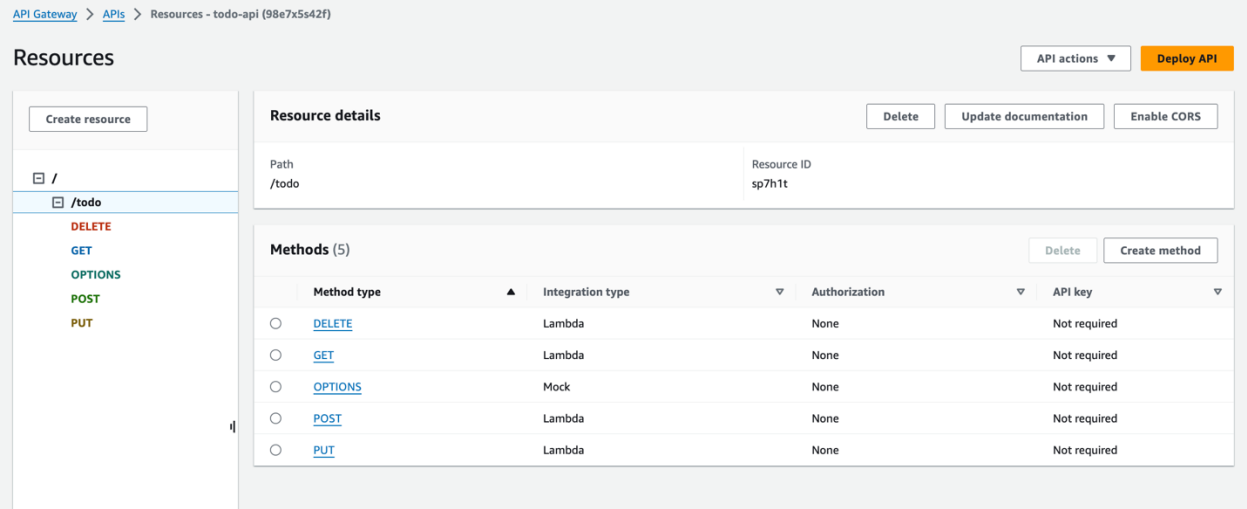
Default timeout

The default timeout is 29 seconds.

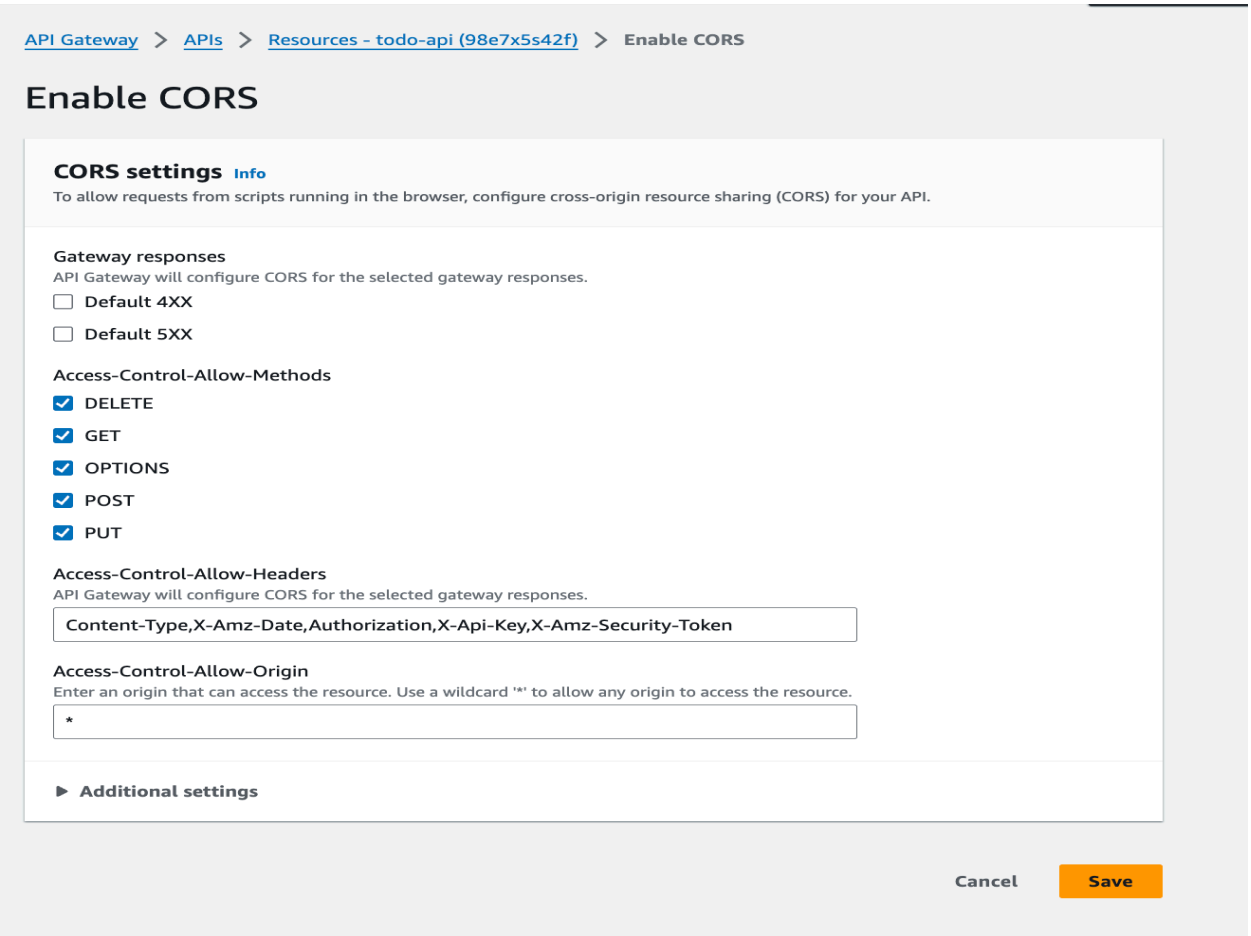
Cancel

Create method

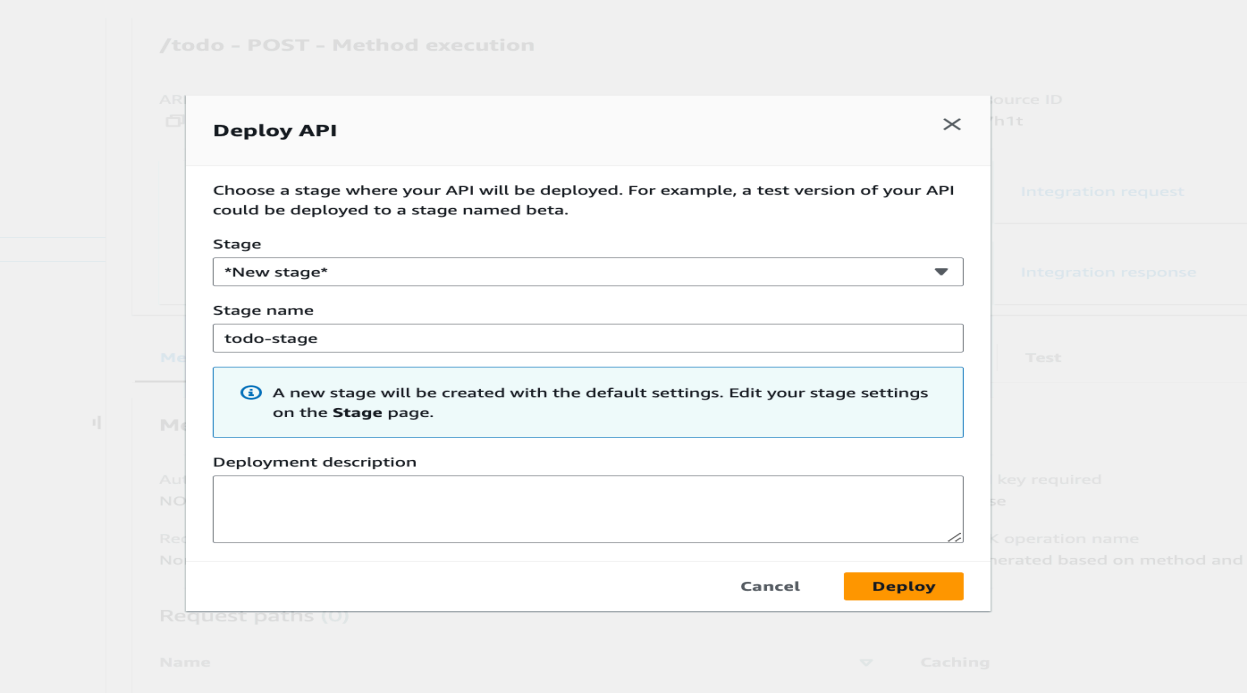
List of created api methods are shown below



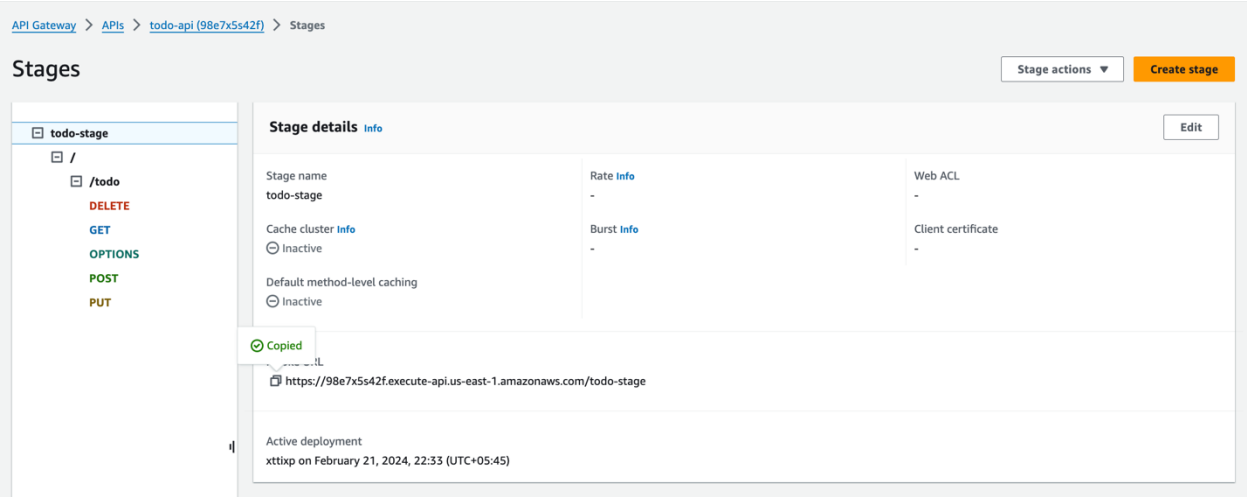
The API CORS is enabled.



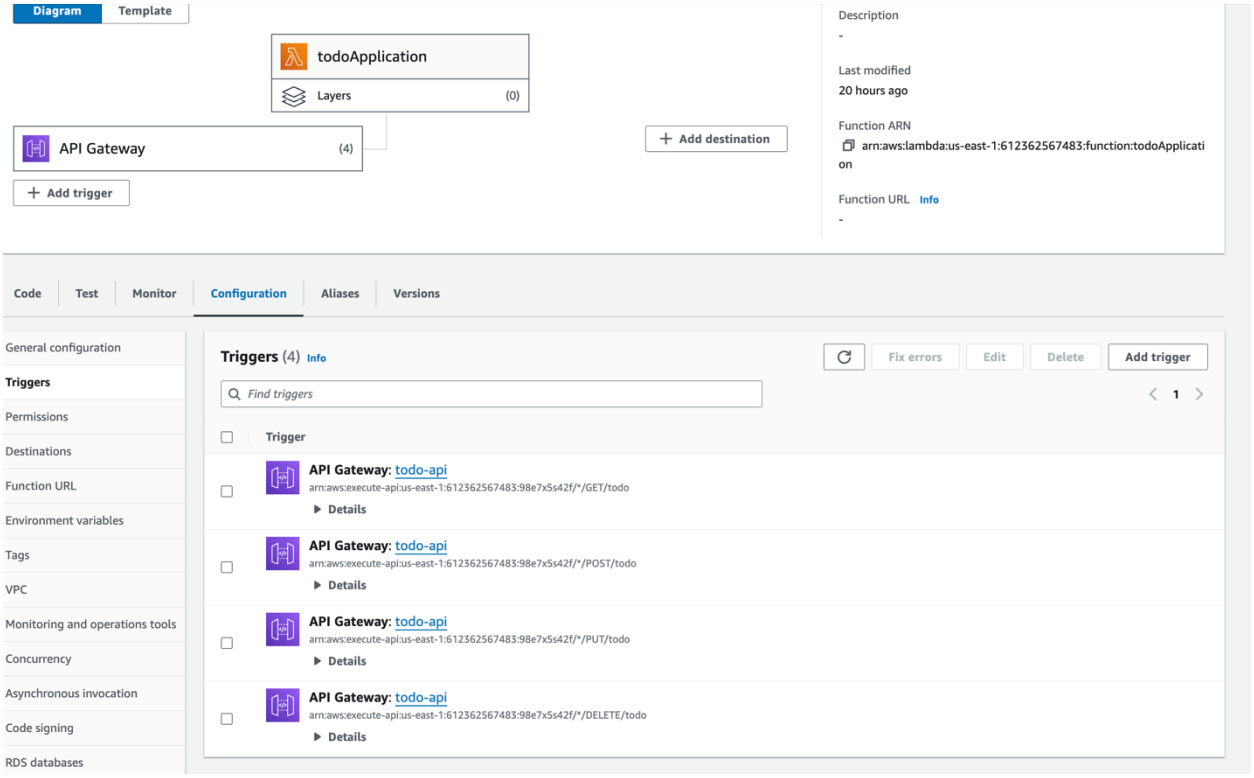
After completion of basic configurations for API gateway, Lambda functions, tables the API is deployed. During the process, new stage is created and deployed.



The invoke url is used to access the API



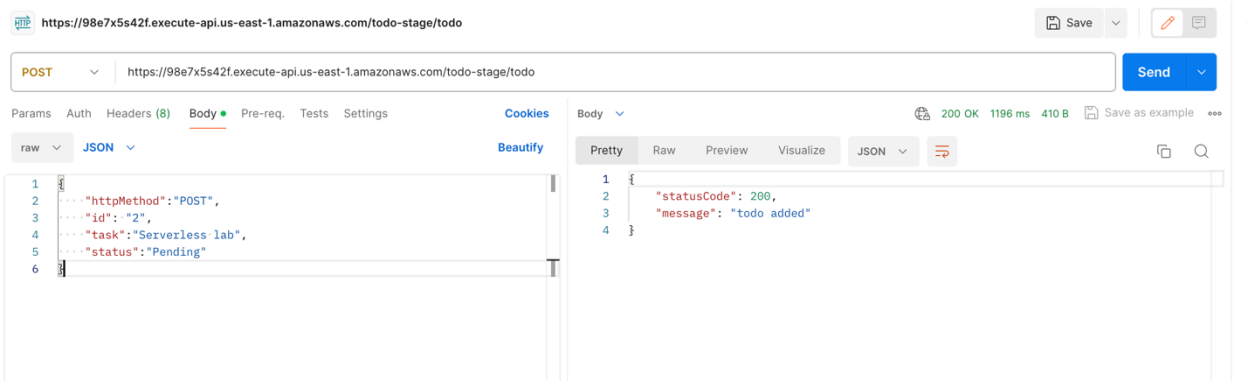
Now, getting back to created lambda function. In the configuration tab the list of triggers of created API Gateway can be viewed.

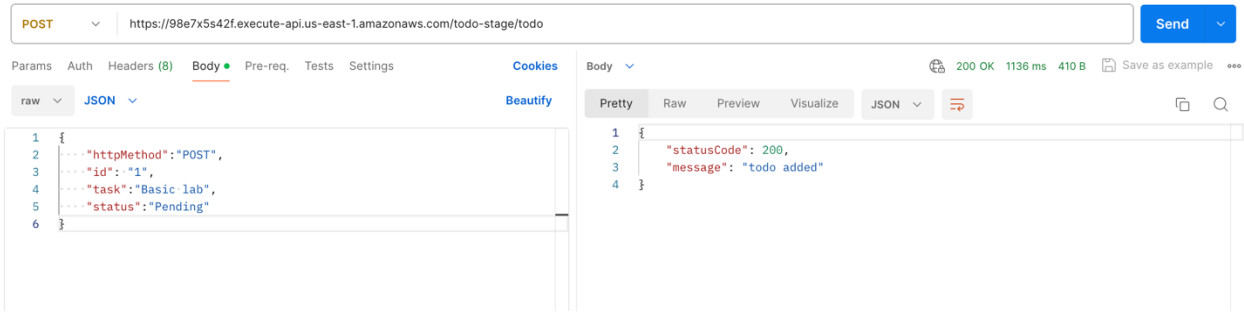


Functions for each request and its operation is created. After that source code is deployed and tested in Postman.

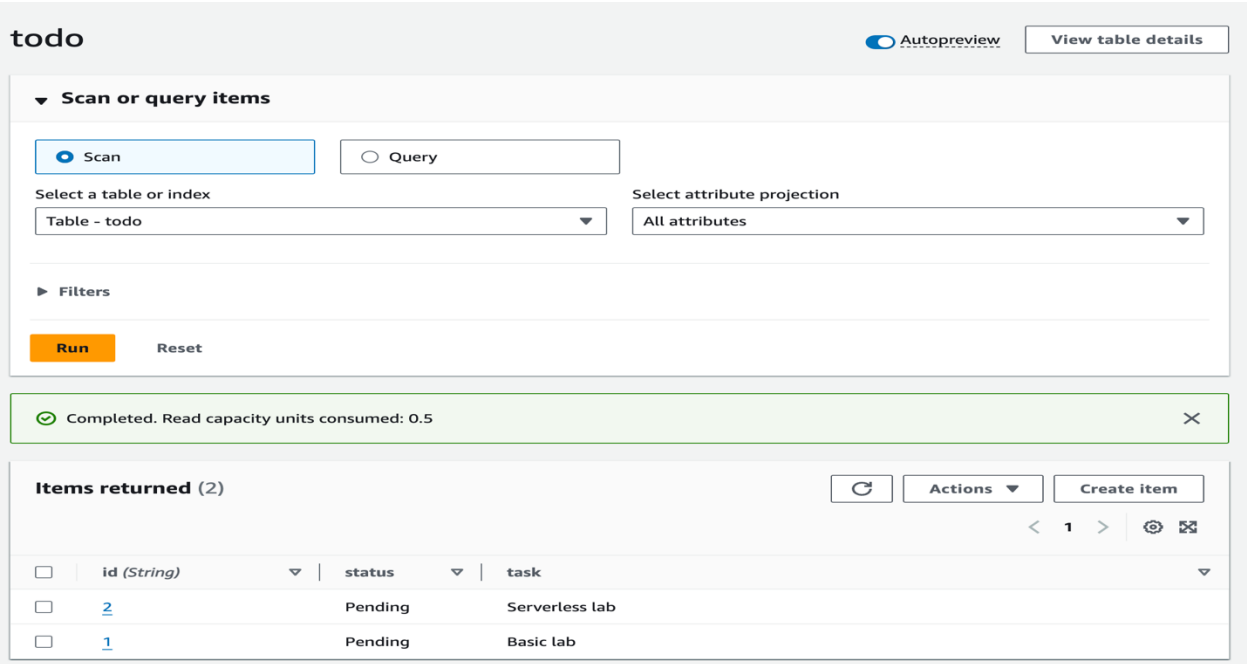
i. POST Request code snippet and postman testing

```
25
26  def post_request(event):
27      try:
28          item={
29              'id':event['id'],
30              'task':event['task'],
31              'status':event['status']
32          }
33          table.put_item(Item=item)
34          response= {
35              "statusCode":200,
36              "message":"todo added"
37          }
38      except ClientError as e:
39          response= {
40              "statusCode":500,
41              "message":"Something went wrong"
42              "data":str(e)
43          }
44      return response
```



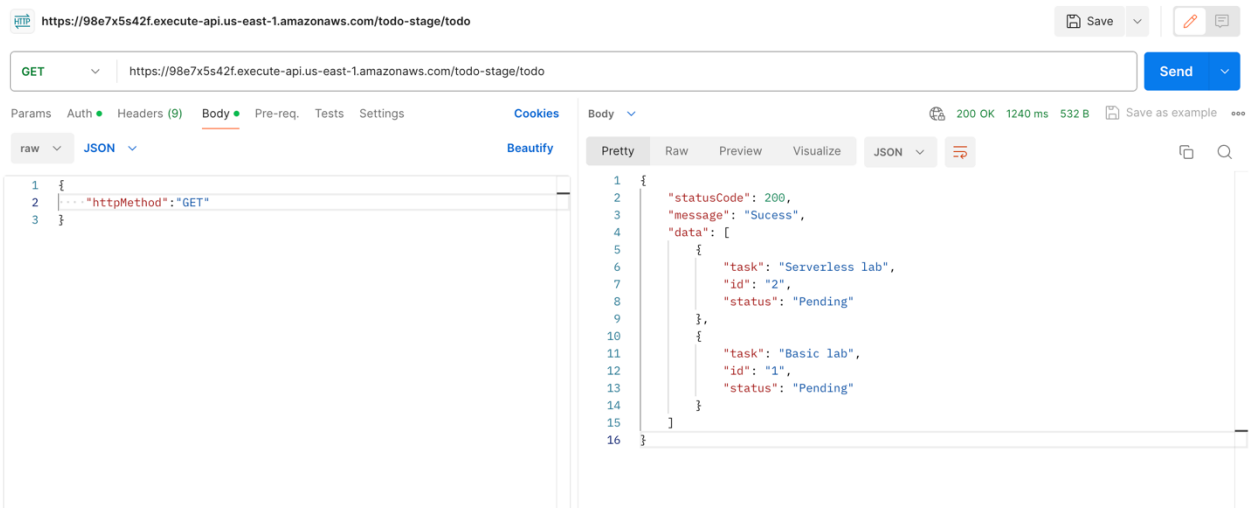


Items list in todo table



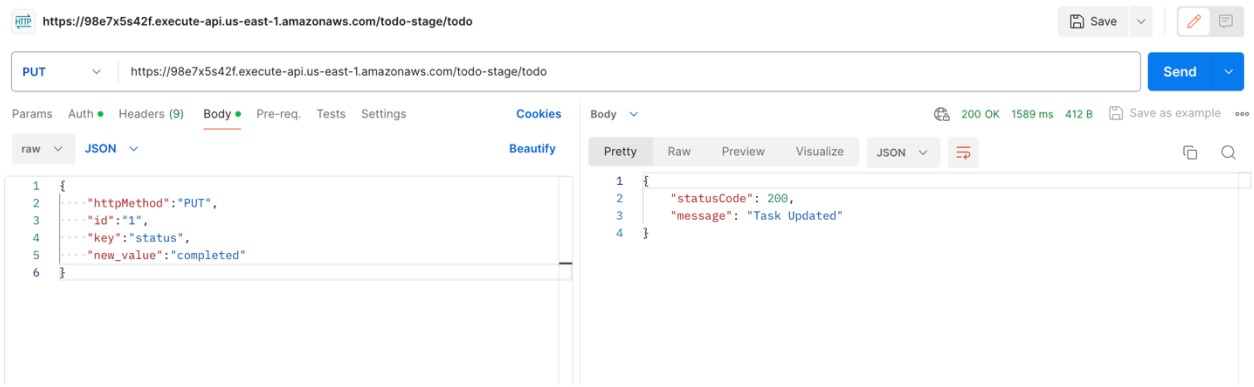
ii. Get request code snippet and postman testing

```python
def get_request(event):
    try:
        data = table.scan()
        items = data.get("Items", [])
        print(items)
        response= {
            "statusCode":200,
            "message":"Sucess",
            "data":items
        }
    except ClientError as e:
        response={
            "statusCode":500,
            "message":"Something went wrong",
            "data":str(e)
        }
    return response
```

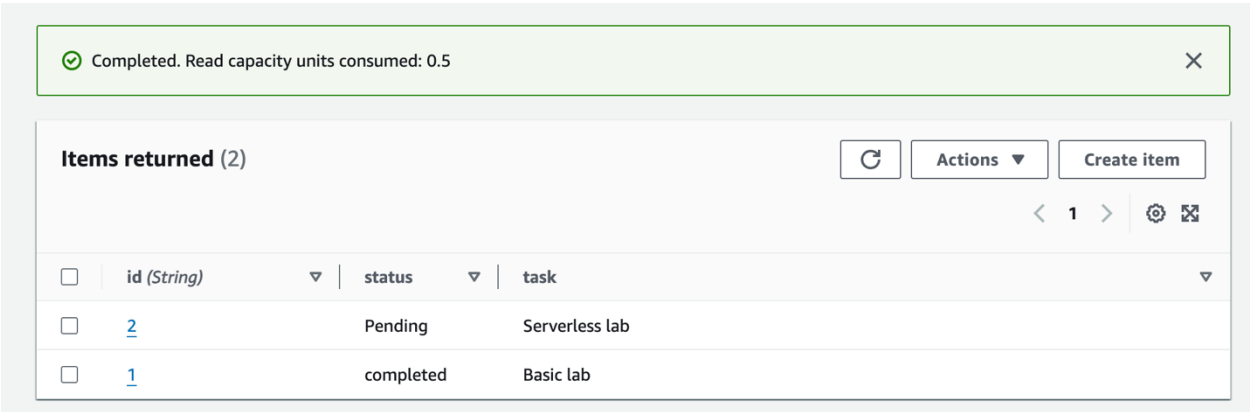


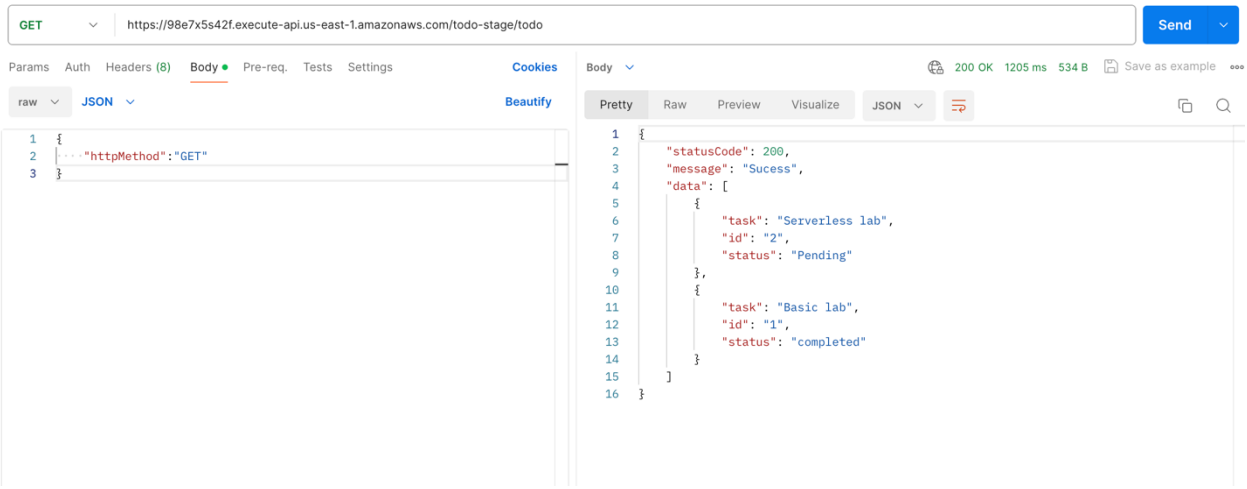
iii. PUT request code snippet and postman testing

```python
def put_request(event):
    try:
        id = event['id']
        update_key = event['key']
        update_value = event['new_value']

        table.update_item(
            Key={'id': id},
            UpdateExpression=f'SET #attr = :value',
            ExpressionAttributeNames={'#attr': update_key},
            ExpressionAttributeValues={':value': update_value},
            ReturnValues='UPDATED_NEW'
        )

        response= {
            "statusCode":200,
            "message":"Task Updated"
        }
    except ClientError as e:
        response= {
            "statusCode":500,
            "message":"Something went wrong",
            "data":str(e)
        }
    return response
```



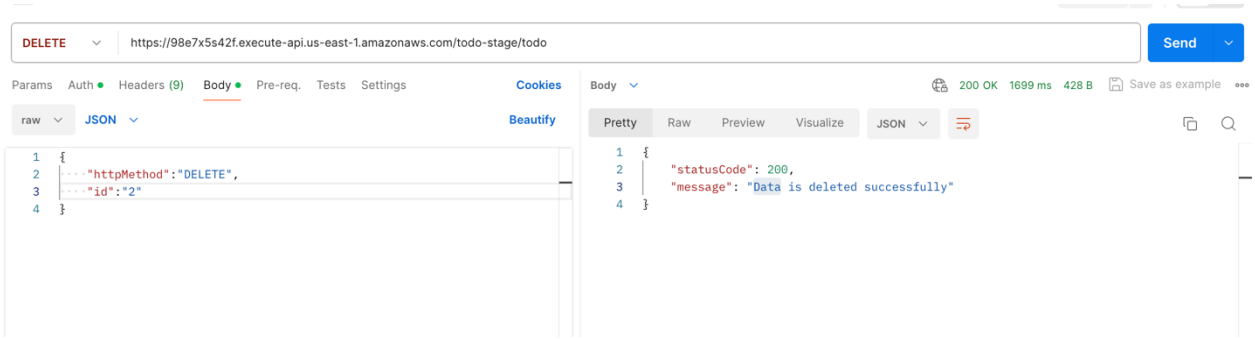
Checking the item data after PUT request



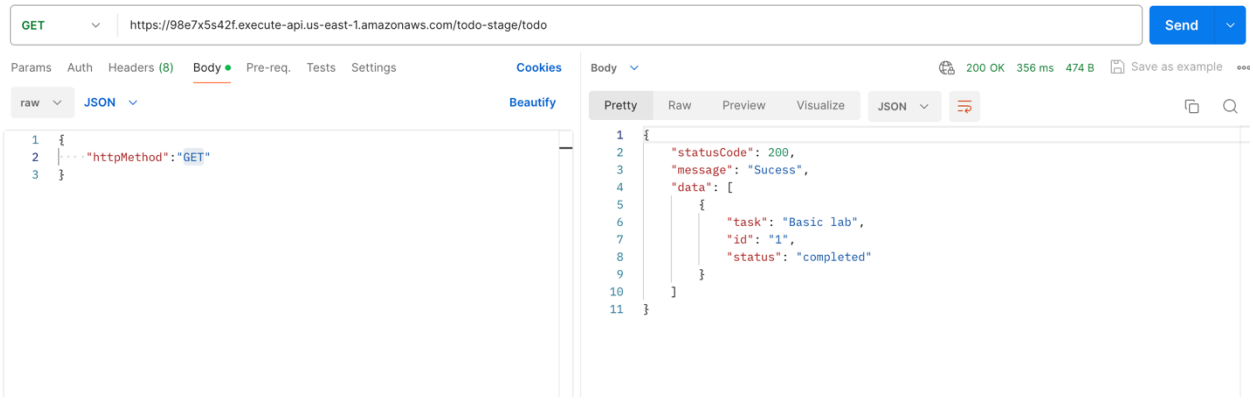


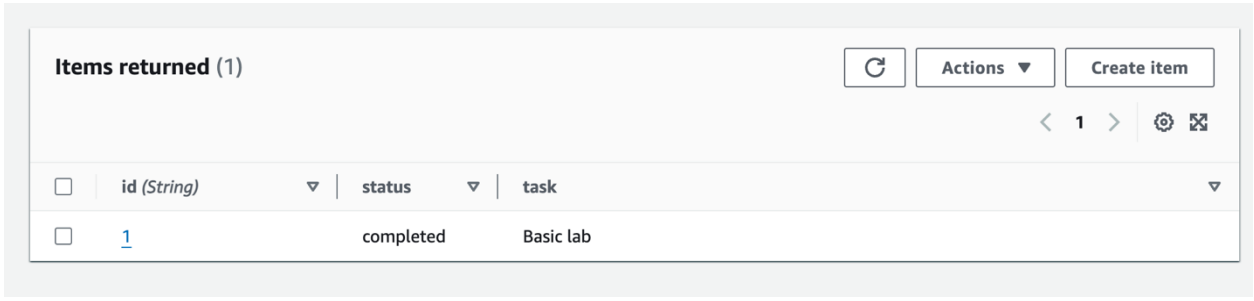
iv. Delete request code snippet and postman testing

```
63
64  def delete_request(event):
65      try:
66          id_value = event['id']   # Get id from event
67          if id_value is None:
68              raise ValueError("ID is missing in the event")
69
70          response = table.delete_item(Key={'id': id_value})
71          return {
72              "statusCode": 200,
73              "message": "Data is deleted successfully",
74          }
75      except Exception as e:
76          return {
77              "statusCode": 500,
78              "message": "An error occurred while deleting data",
79              "error": str(e)
80          }
81
```



Checking the list of items after deleting in postman and items of table





3. Serverless Data Processing Pipeline

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

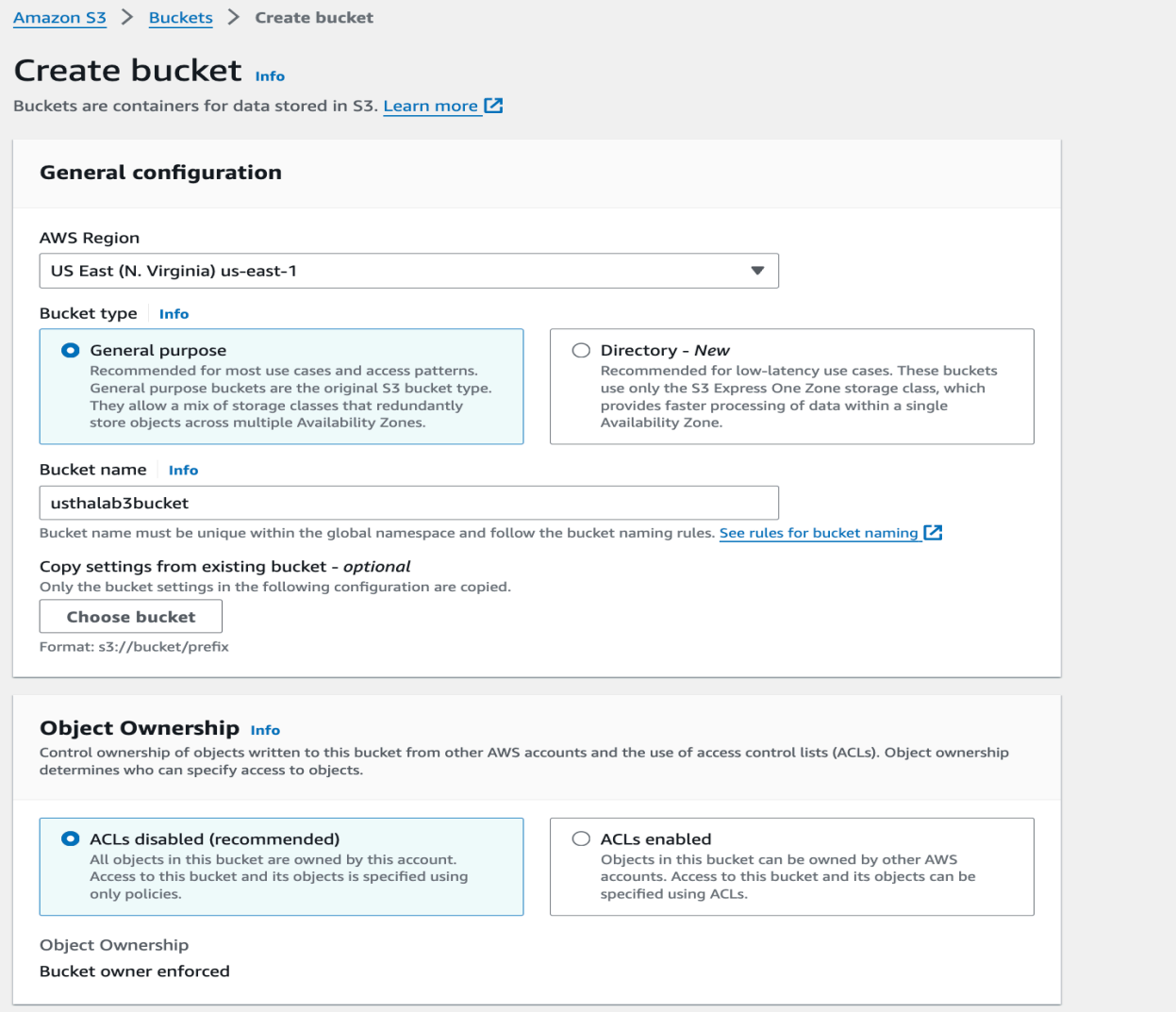
Approach:

- **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.

3.1. S3 bucket creation

Creating a two different S3 buckets with required configurations



Amazon S3 > Buckets > Create bucket

Create bucket Info

Buckets are containers for data stored in S3. Learn more

General configuration

AWS Region

US East (N. Virginia) us-east-1

Bucket type Info

- (•) General purpose
  Recommended for most use cases and access patterns. General purpose buckets are the original S3 bucket type. They allow a mix of storage classes that redundantly store objects across multiple Availability Zones.
- ( ) Directory - *New*
  Recommended for low-latency use cases. These buckets use only the S3 Express One Zone storage class, which provides faster processing of data within a single Availability Zone.

Bucket name Info

utshalab3bucket2

Bucket name must be unique within the global namespace and follow the bucket naming rules. See rules for bucket naming

Copy settings from existing bucket - *optional*

Only the bucket settings in the following configuration are copied.

Choose bucket

Format: s3://bucket/prefix

Object Ownership Info

Control ownership of objects written to this bucket from other AWS accounts and the use of access control lists (ACLs). Object ownership determines who can specify access to objects.

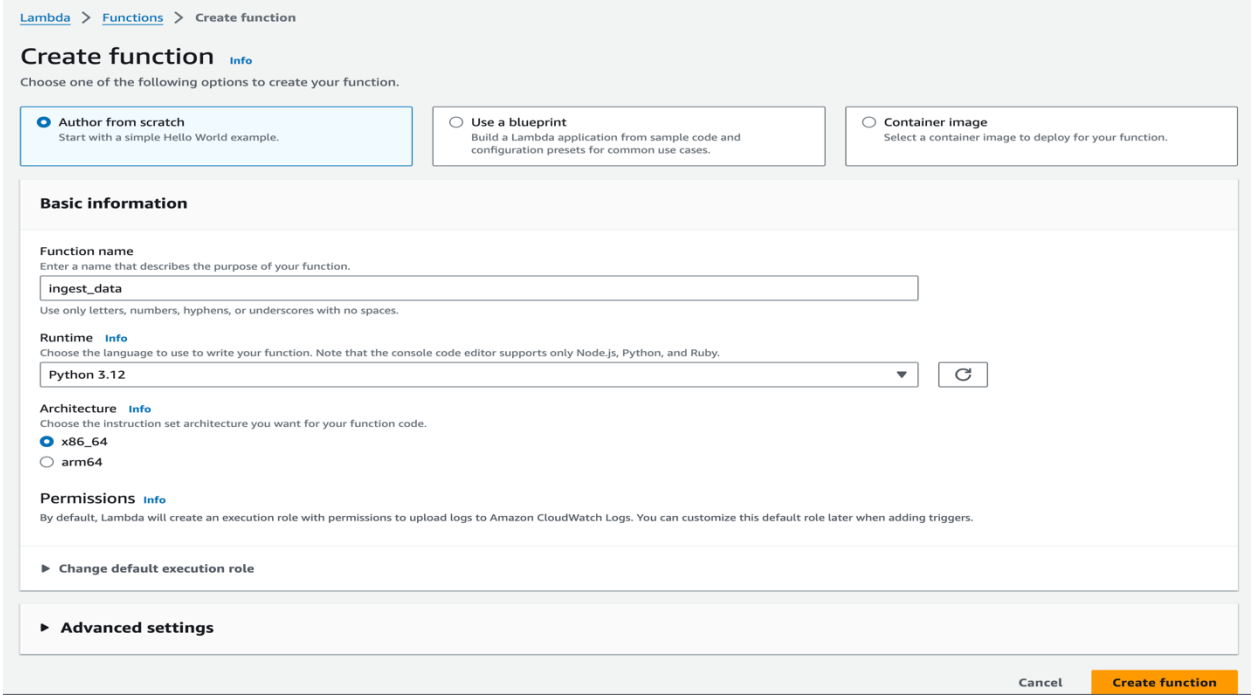
- (•) ACLs disabled (recommended)
  All objects in this bucket are owned by this account. Access to this bucket and its objects is specified using only policies.
- ( ) ACLs enabled
  Objects in this bucket can be owned by other AWS accounts. Access to this bucket and its objects can be specified using ACLs.

Object Ownership

Bucket owner enforced

3.2. Lambda function implementaion

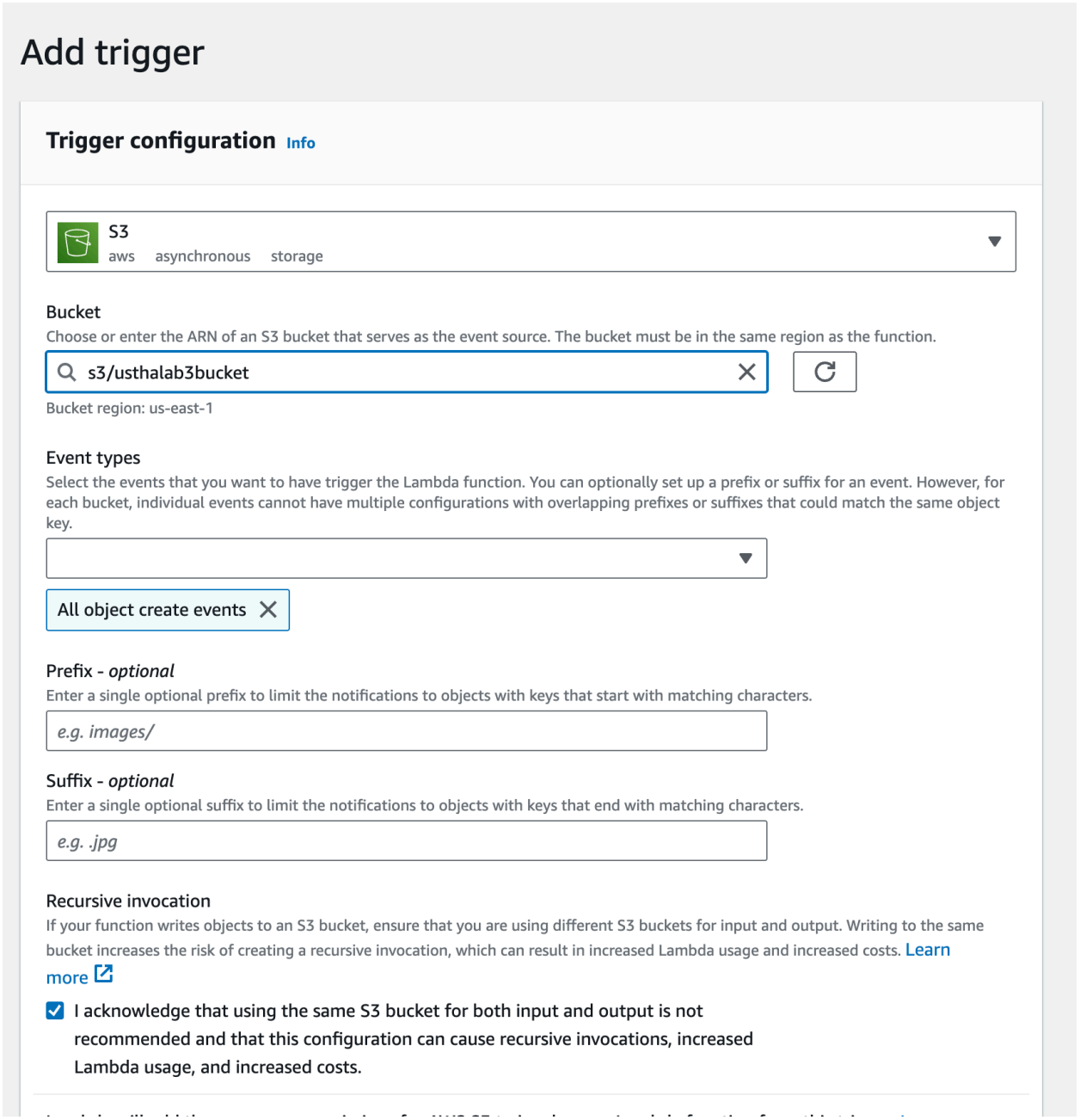
Create a lambda function with runtime Python3.12



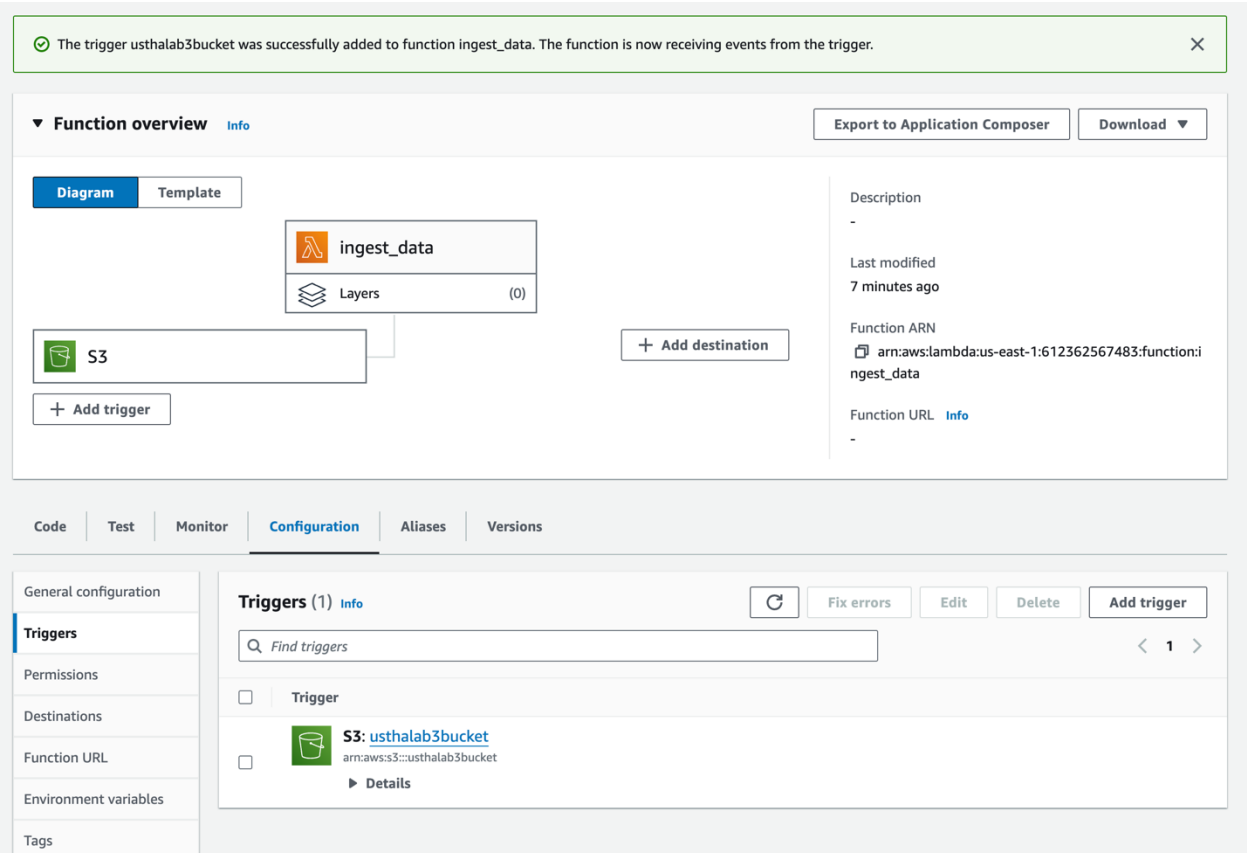
Below code snippet is the implementation of ingested data in lambda function. After lambda function implementation, the code is deployed.

```python
import json
import boto3

s3 = boto3.client('s3')

def lambda_handler(event, context):
    # Get the bucket names from the event
    source_bucket = event['Records'][0]['s3']['bucket']['name']
    key = event['Records'][0]['s3']['object']['key']

    # Download the file from the source bucket
    response = s3.get_object(Bucket=source_bucket, Key=key)
    content = response['Body'].read().decode('utf-8')

    # Convert content to uppercase
    uppercase_content = content.upper()

    # Upload the uppercase content to the next bucket
    uppercase_bucket = "utshalab3bucket2"
    s3.put_object(Body=uppercase_content, Bucket=uppercase_bucket, Key=key)

    # Return a response
    return {
        'statusCode': 200,
        'body': json.dumps('Data of the file transformed and uploaded successfully')
    }

```

Adding trigger in the created lambda function with required configurations. So, that whenever a text file is uploaded triggered S3 bucket, its processed the data of the text file and uploads in specified S3 bucket location.

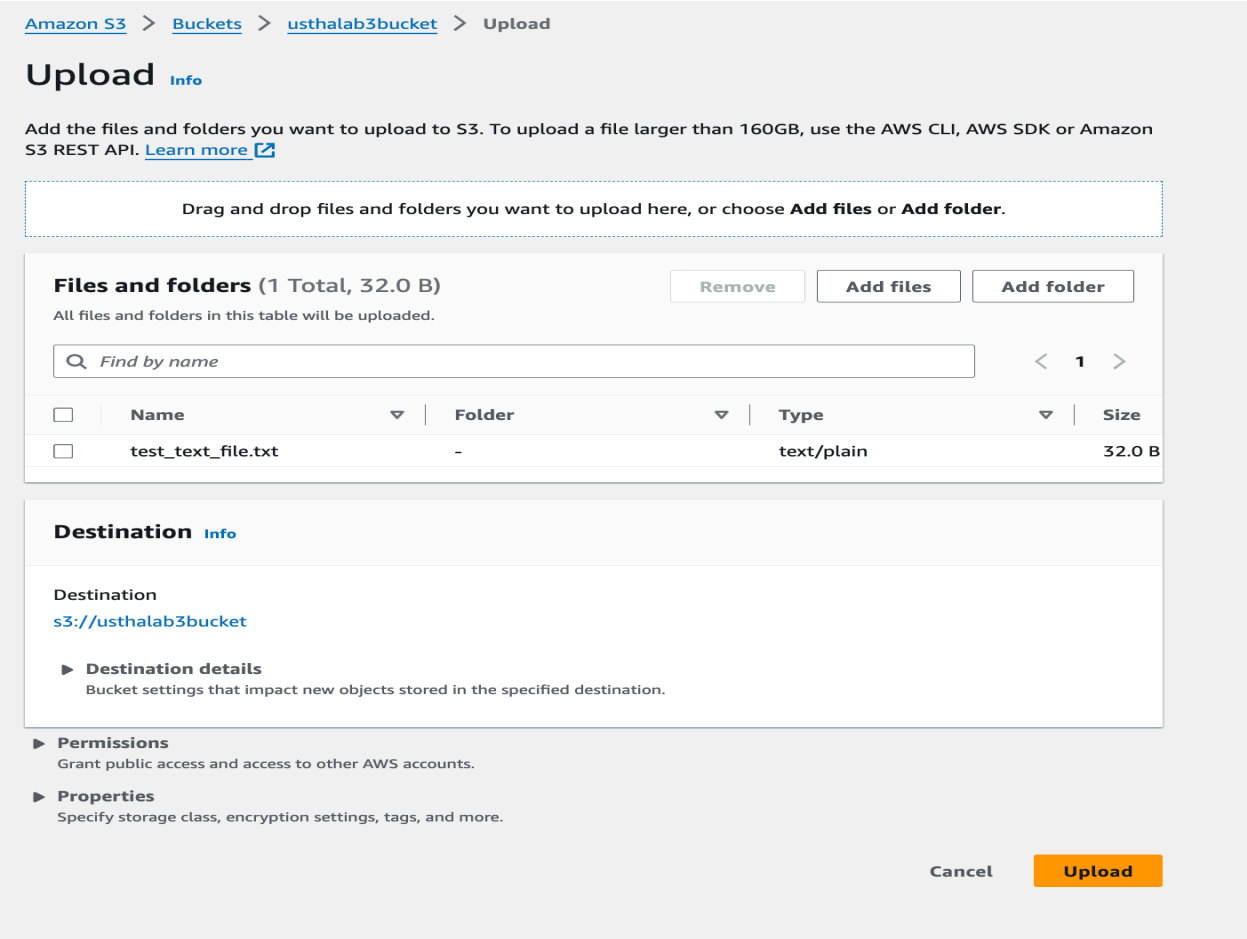


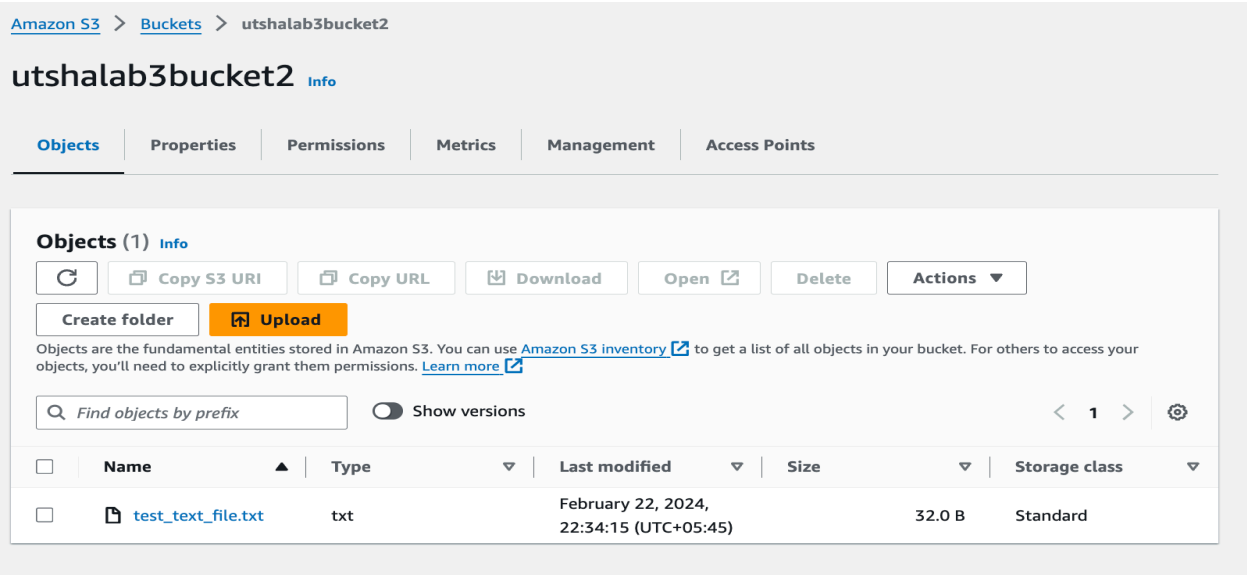
The created S3 trigger can be viewed.



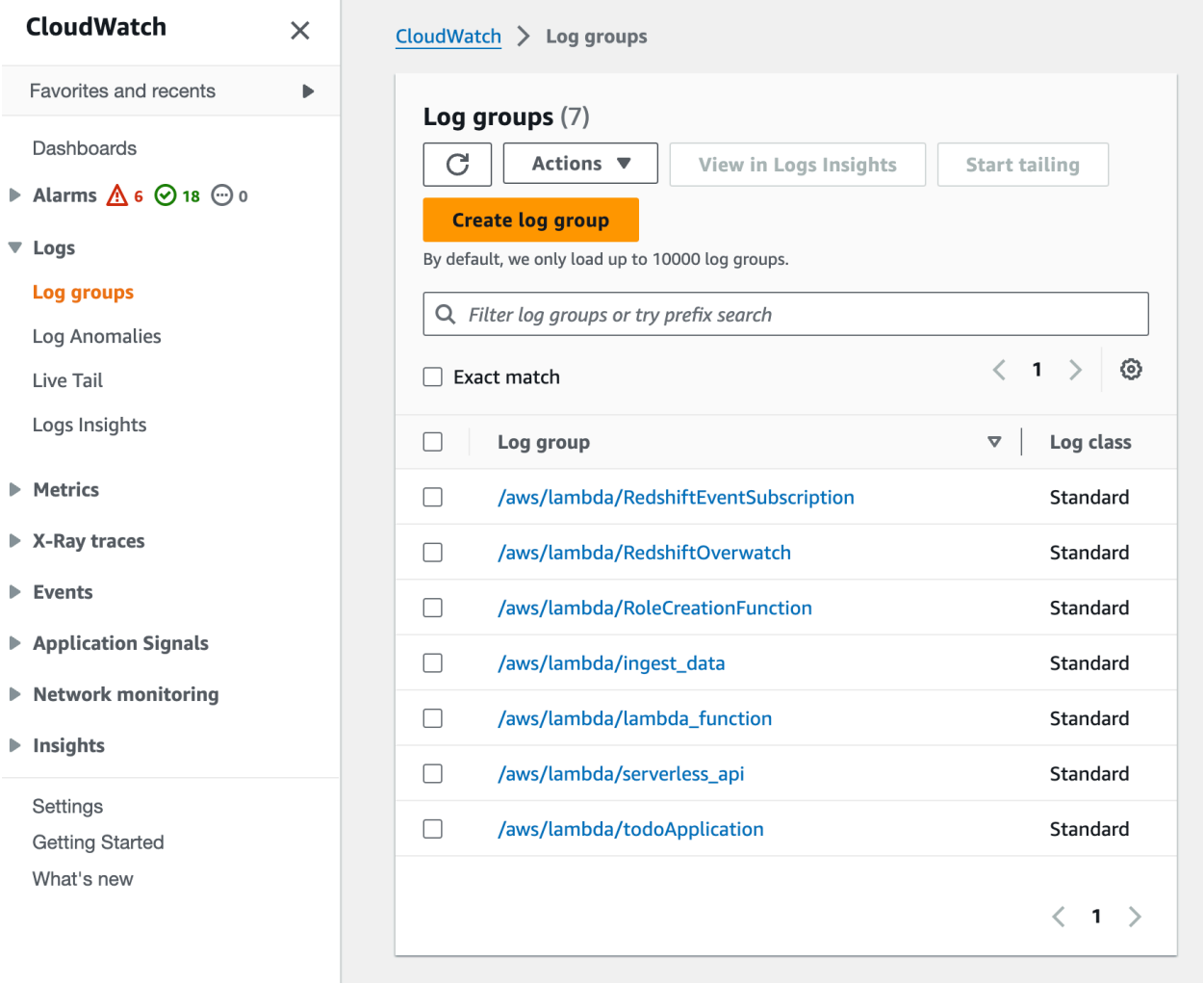
3.3. Testing the implementation and monitoring in cloud watch logs

A text file with some content is uploaded in a S3 bucket.

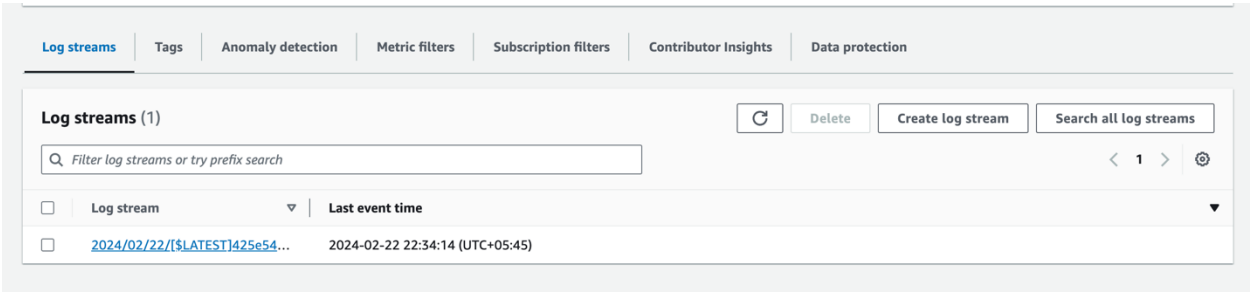




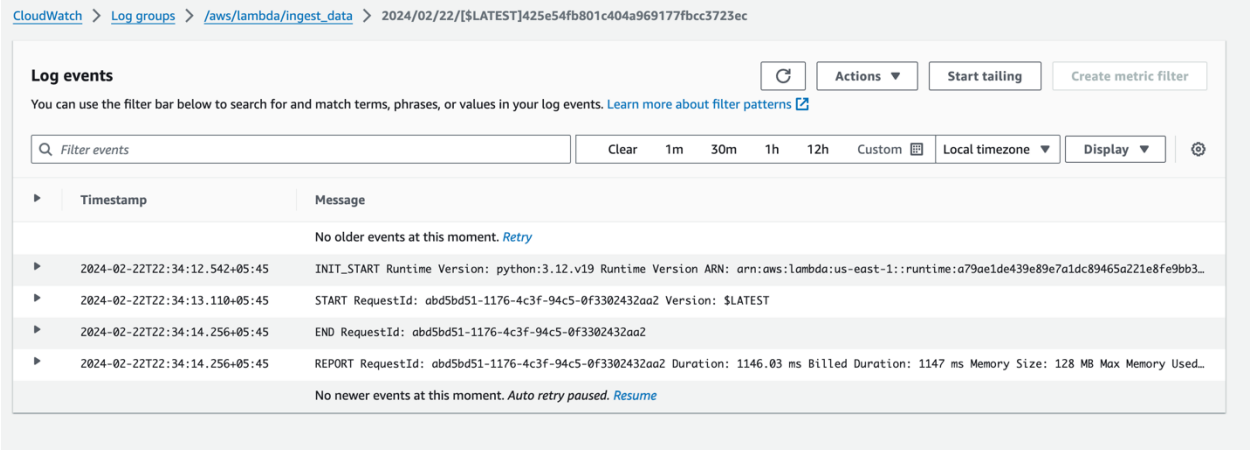
Now, navigating to Cloud watch and checking the log groups. The newly created lambda function ‘ingest_data’ is in the list of log group.



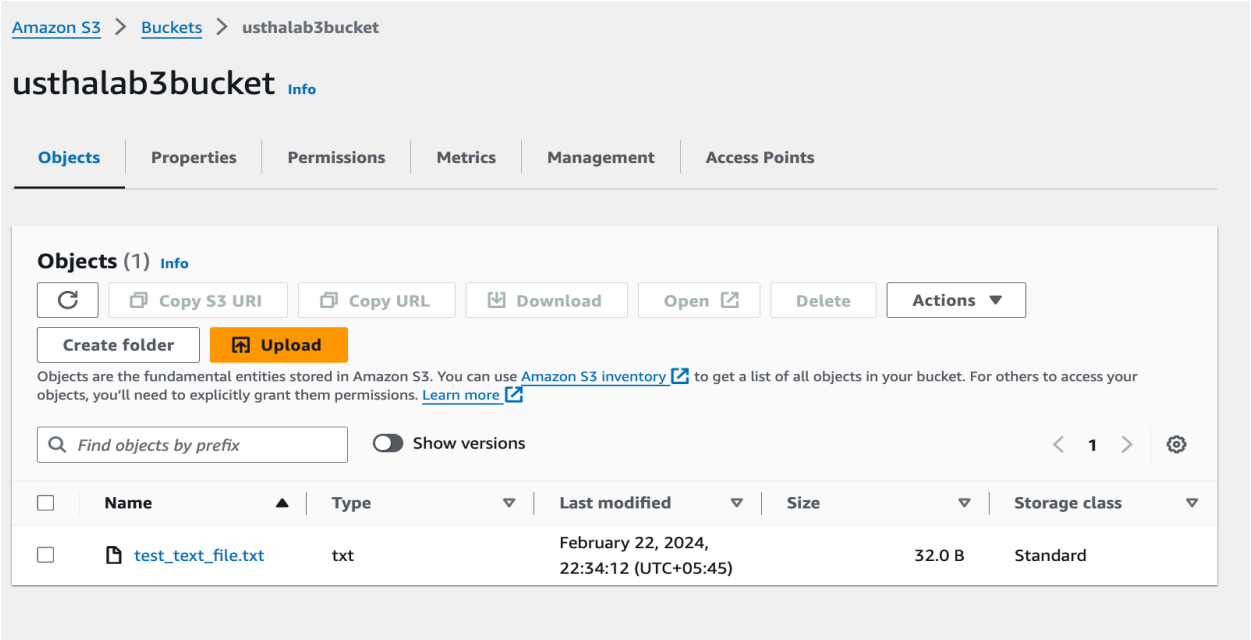
Opening the log group and in Log streams, there will be the list of the logs streams of the lambda functions.



Based upon the log events, the implementation perfomed has been executed sucessfully.



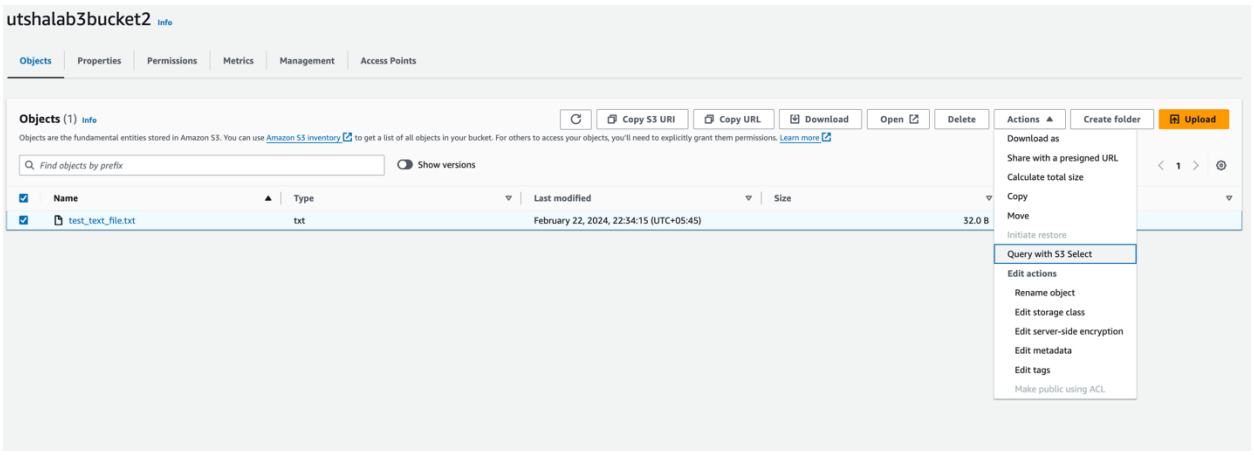
Now, while checking in the S3 bucket location where processed data is saved, ulploaded text file can viewed.

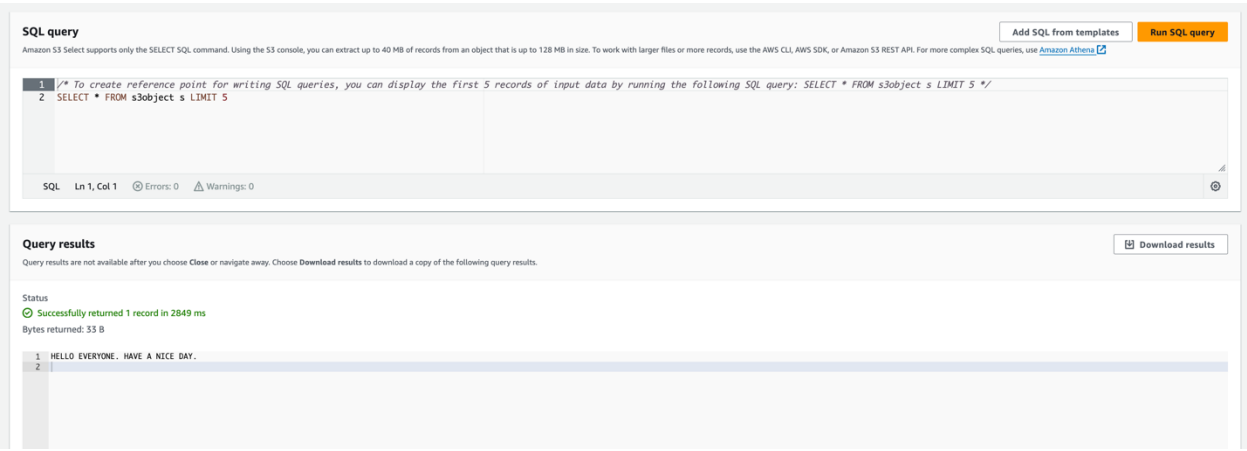


Initially text file content were in lower case.

Now, the content of the text file are in uppercase.

The content of the file can be also viewed from “Query with S3 select”.




SQL query
Add SQL from templates
Run SQL query
Amazon S3 Select supports only the SELECT SQL command. Using the S3 console, you can extract up to 40 MB of records from an object that is up to 128 MB in size. To work with larger files or more records, use the AWS CLI, AWS SDK, or Amazon S3 REST API. For more complex SQL queries, use Amazon Athena
1 /* To create reference point for writing SQL queries, you can display the first 5 records of input data by running the following SQL query: SELECT * FROM s3object s LIMIT 5 */
2 SELECT * FROM s3object s LIMIT 5
SQL
Ln 1, Col 1
Errors: 0
Warnings: 0
Query results
Download results
Query results are not available after you choose Close or navigate away. Choose Download results to download a copy of the following query results.
Status
Successfully returned 1 record in 2849 ms
Bytes returned: 33 B
1 HELLO EVERYONE. HAVE A NICE DAY.
2