AWS Project : Serverless web app

Documentation

By : Abouchiba Mohamed Yassir

To see the files used in this project, go to this link:
https://github.com/M-Yassir/AWS-Projects/tree/main/Serverless_Web_App

Introduction

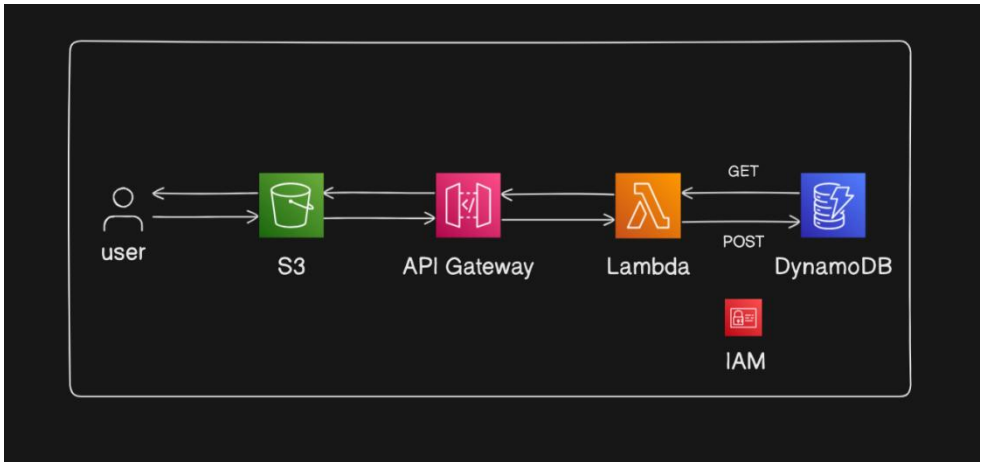
Project Description

This project demonstrates the development of a serverless web application for user registration and management using Amazon Web Services (AWS). The application allows users to submit their details (username, password, email, and phone number) through a web form and retrieves all registered users for display. Built entirely within the AWS Free Tier, it showcases how serverless architecture can solve common challenges like scalability, cost-efficiency, and ease of deployment.

Key Features

- User Registration: A web form to collect and store user data.
- User Retrieval: A button to fetch and display all registered users.
- Serverless Architecture: Uses AWS Lambda, API Gateway, and DynamoDB for backend functionality.
- Static Website Hosting: The frontend is hosted on Amazon S3.

AWS architecture diagram



Step-by-Step implementation

NB: - All the necessary code is on the repository given above.
- The creation of the index.html file isn't covered here as it isn't our topic.

Step1: Creating a DynamoDB table and Lambda functions:

First we go to DynamoDB service then we click on '*Create table*'

We fill in the necessary informations like it is shown below:

Create table

Table details Info

DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

Table name

This will be used to identify your table.

Users

Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.).

Partition key

The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.

username | String

1 to 255 characters and case sensitive.

Sort key - *optional*

You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.

Enter the sort key name | String

1 to 255 characters and case sensitive.

Table settings

Default settings
The fastest way to create your table. You can modify most of these settings after your table has been created. To modify these settings now, choose 'Customize settings'.

Customize settings
Use these advanced features to make DynamoDB work better for your needs.

Default table settings

These are the default settings for your new table. You can change some of these settings after creating the table.

Setting	Value	Editable after creation
Table class	DynamoDB Standard	Yes
Capacity mode	On-demand	Yes
Maximum read capacity units	-	Yes
Maximum write capacity units	-	Yes
Local secondary indexes	-	No
Global secondary indexes	-	Yes
Encryption key management	Owned by Amazon DynamoDB	Yes
Deletion protection	Off	Yes
Resource-based policy	Not active	Yes

Tags

Tags are pairs of keys and optional values, that you can assign to AWS resources. You can use tags to control access to your resources or track your AWS spending.

No tags are associated with the resource.

Add new tag

You can add 50 more tags.

Cancel | Create table

We click on '*Create table*'

We go to IAM to create a role to let the lambda function access DynamoDB:

IAM > Roles > Create role

Step 1
Select trusted entity
Step 2
Add permissions
Step 3
Name, review, and create

Select trusted entity Info

Trusted entity type

- AWS service — Allow AWS services like EC2, Lambda, or others to perform actions in this account.
- AWS account — Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.
- Web identity — Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.
- SAML 2.0 federation — Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.
- Custom trust policy — Create a custom trust policy to enable others to perform actions in this account.

Use case

Allow an AWS service like EC2, Lambda, or others to perform actions in this account.

Service or use case

Lambda

Choose a use case for the specified service.

Use case

Lambda
Allows Lambda functions to call AWS services on your behalf.

Cancel | Next

IAM > Roles > Create role

Step 1
Select trusted entity
Step 2
Add permissions
Step 3
Name, review, and create

Add permissions Info

Permissions policies (1/1025) Info

Choose one or more policies to attach to your new role.

Filter by Type: All types

dynam — 5 matches

Policy name	Type	Description
AmazonDynamoDBFullAccess	AWS managed	Provides full access to Amazon DynamoD...
AmazonDynamoDBFullAccesswithDataPipeline	AWS managed	This policy is on a deprecation path. See ...
AmazonDynamoDBReadOnlyAccess	AWS managed	Provides read only access to Amazon Dyn...
AWSLambdaDynamoDBExecutionRole	AWS managed	Provides list and read access to DynamoD...
AWSLambdaInvocation-DynamoDB	AWS managed	Provides read access to DynamoDB Strea...

Set permissions boundary - *optional*

Cancel | Previous | Next

Step 1
Select trusted entity
Step 2
Add permissions
Step 3
Name, review, and create

Name, review, and create

Role details

Role name
Enter a meaningful name to identify this role.
lambdaDynamoDB-role
Maximum 64 characters. Use alphanumeric and '+=,.@-_' characters.

Description
Add a short explanation for this role.
Allows Lambda functions to acess to DynamoDB.
Maximum 1000 characters. Use letters (A-Z and a-z), numbers (0-9), tabs, new lines, or any of the following characters: _+=,.@-/\[{}]#$%,^*@;:"'

Step 1: Select trusted entities

Edit

Trust policy

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sts:AssumeRole"
            ],
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            }
        }
    ]
}
```

Step 2: Add permissions

Edit

Permissions policy summary

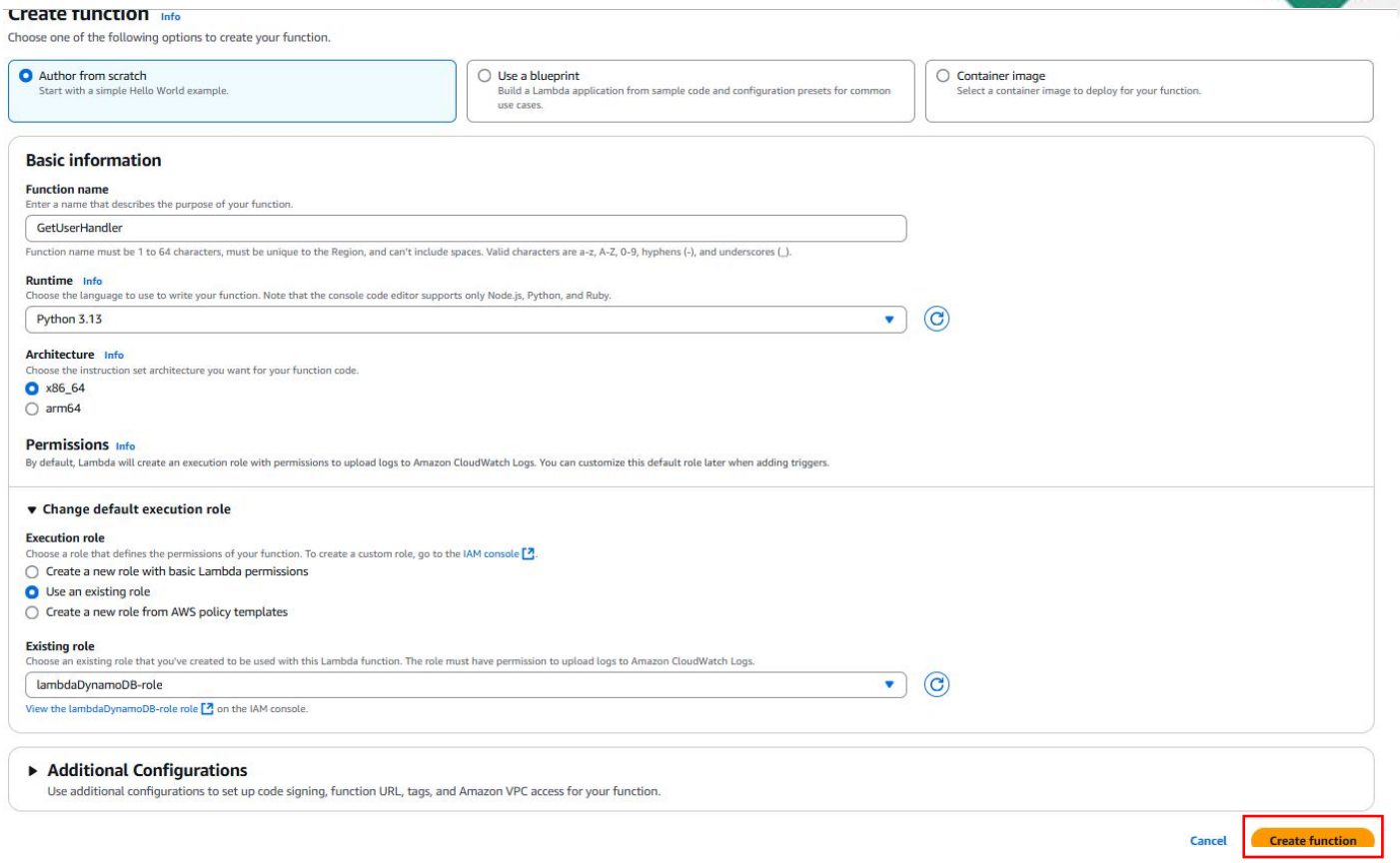
Policy name	Type	Attached as
AmazonDynamoDBFullAccess	AWS managed	Permissions policy

Step 3: Add tags

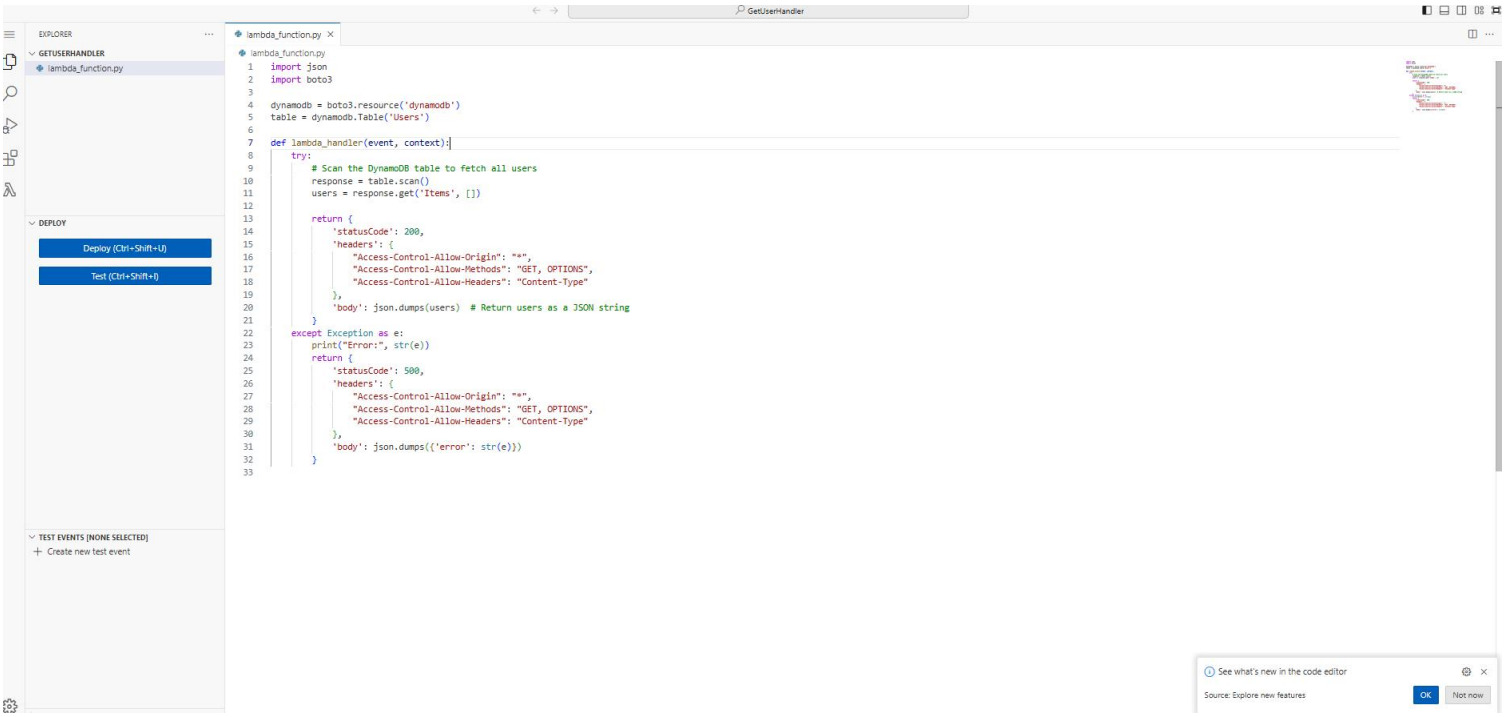
Add tags - *optional* Info
Tags are key-value pairs that you can add to AWS resources to help identify, organize, or search for resources.
No tags associated with the resource.
Add new tag
You can add up to 50 more tags.

Cancel | Previous | Create role

Then, we go to the Lambda service and we create a function to get user's informations:



We click on 'create function', after that, we go to 'code source' and we type the following python code (the code is on the repository):



We create another function to register users, the same steps are here:

Choose one of the following options to create your function.

- (●) **Author from scratch** — Start with a simple Hello World example.
- ( ) **Use a blueprint** — Build a Lambda application from sample code and configuration presets for common use cases.
- ( ) **Container image** — Select a container image to deploy for your function.

Basic information

Function name
Enter a name that describes the purpose of your function.

RegisterUserHandler

Function name must be 1 to 64 characters, must be unique to the Region, and can't include spaces. Valid characters are a-z, A-Z, 0-9, hyphens (-), and underscores (_).

Runtime Info
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.

Python 3.13

Architecture Info
Choose the instruction set architecture you want for your function code.

- (●) x86_64
- ( ) arm64

Permissions Info
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

▼ **Change default execution role**

Execution role
Choose a role that defines the permissions of your function. To create a custom role, go to the IAM console.

- ( ) Create a new role with basic Lambda permissions
- (●) Use an existing role
- ( ) Create a new role from AWS policy templates

Existing role
Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.

lambdaDynamoDB-role

View the lambdaDynamoDB-role role on the IAM console.

▶ **Additional Configurations**
Use additional configurations to set up code signing, function URL, tags, and Amazon VPC access for your function.

Cancel | Create function

We click on 'create function', after that, we go to 'code source' and we type the following python code (the code is on the repository):

EXPLORER
REGISTERUSERHANDLER
lambda_function.py

DEPLOY
Deploy (Ctrl+Shift+U)
Test (Ctrl+Shift+I)

TEST EVENTS [SELECTED: (UNSAVED) TEST EVENT]
+ Create new test event

ENVIRONMENT VARIABLES

lambda_function.py

```python
import json
import boto3

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

def lambda_handler(event, context):
    print("Received event:", json.dumps(event))  # Log the entire event

    try:
        # Use the event directly (no need to extract 'body')
        body = event
        print("Parsed body:", body)  # Log the parsed body

        # Check for required fields
        required_fields = ['username', 'password', 'email', 'phone']
        if not all(field in body for field in required_fields):
            print("Missing required fields:", required_fields)  # Log missing fields
            return {
                'statusCode': 400,
                'headers': {
                    "Access-Control-Allow-Origin": "*",
                    "Access-Control-Allow-Methods": "POST, GET, OPTIONS",
                    "Access-Control-Allow-Headers": "Content-Type"
                },
                'body': json.dumps({'error': 'Missing required fields'})
            }

        # Insert the user into DynamoDB
        table.put_item(Item=body)
        print("User inserted successfully:", body)  # Log successful insertion

        return {
            'statusCode': 200,
            'headers': {
                "Access-Control-Allow-Origin": "*",
                "Access-Control-Allow-Methods": "POST, GET, OPTIONS",
                "Access-Control-Allow-Headers": "Content-Type"
            },
            'body': json.dumps({'message': 'User registered successfully!'})
        }
    except Exception as e:
        print("Error:", str(e))  # Log any errors
        return {
            'statusCode': 500,
            'headers': {
                "Access-Control-Allow-Origin": "*",
                "Access-Control-Allow-Methods": "POST, GET, OPTIONS",
                "Access-Control-Allow-Headers": "Content-Type"
            },
            'body': json.dumps({'error': str(e)})
        }
```

Step2: Creating an API to trigger our lambda function using API gateway:

First we go to API gateway service, and we create a REST API

Choose an API type Info

HTTP API

Build low-latency and cost-effective REST APIs with built-in features such as OIDC and OAuth2, and native CORS support.

Works with the following:
Lambda, HTTP backends

Import | Build

WebSocket API

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

Works with the following:
Lambda, HTTP, AWS Services

Build

REST API

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

Works with the following:
Lambda, HTTP, AWS Services

Import | Build

Create REST API Info 1

API details

- New API — Create a new REST API.
- Clone existing API — Create a copy of an API in this AWS account.
- Import API — Import an API from an OpenAPI definition.
- Example API — Learn about API Gateway with an example API.

API name
user

Description - *optional*

API endpoint type
Regional APIs are deployed in the current AWS Region. Edge-optimized APIs route requests to the nearest CloudFront Point of Presence. Private APIs are only accessible from VPCs.
Edge-optimized

Cancel | Create API

We create a GET and POST methods

Create method

Method details

Method type
GET

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.

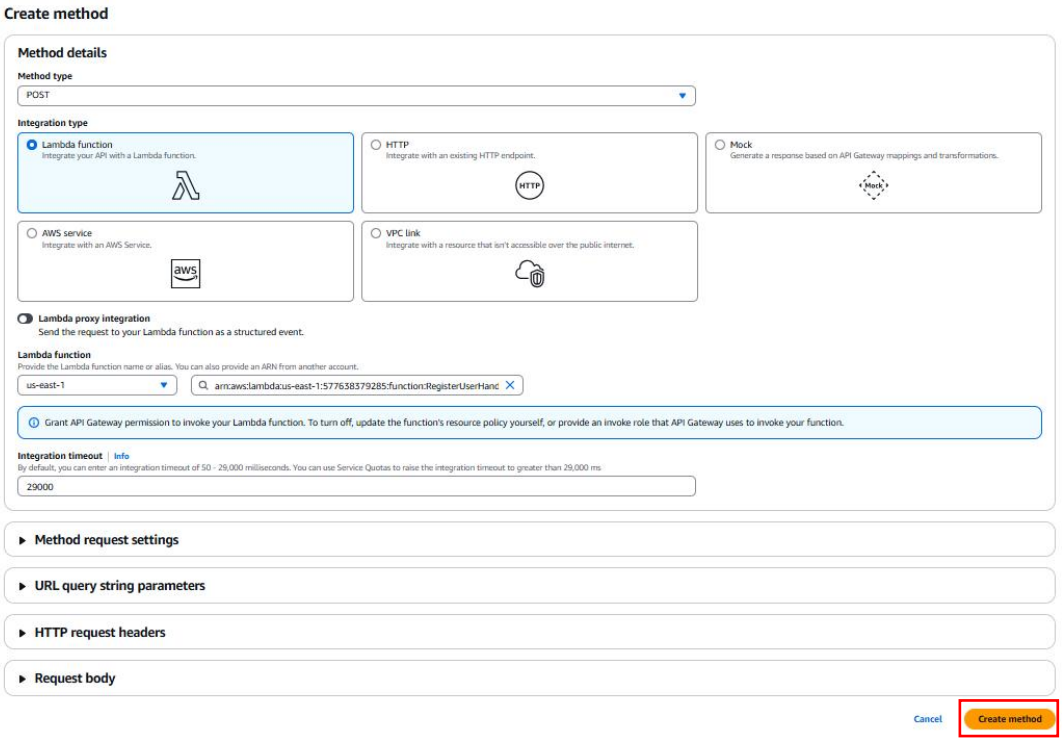
Lambda function
Provide the Lambda function name or alias. You can also provide an ARN from another account.
us-east-1 | arn:aws:lambda:us-east-1:577638379285:function:GetUserHandler

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.

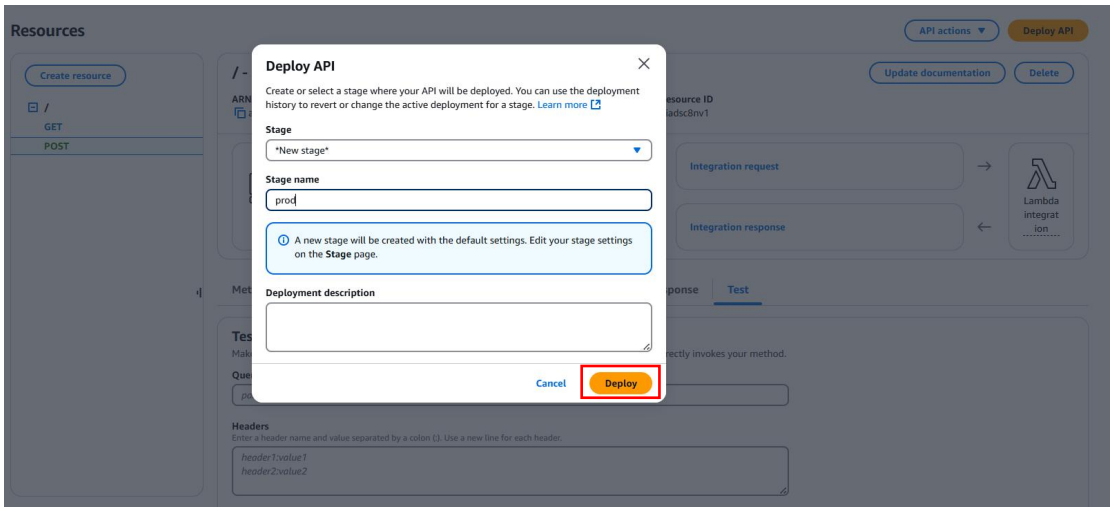
Integration timeout | Info
By default, you can enter an integration timeout of 50 - 29,000 milliseconds. You can use Service Quotas to raise the integration timeout to greater than 29,000 ms.
29000

- Method request settings
- URL query string parameters
- HTTP request headers
- Request body

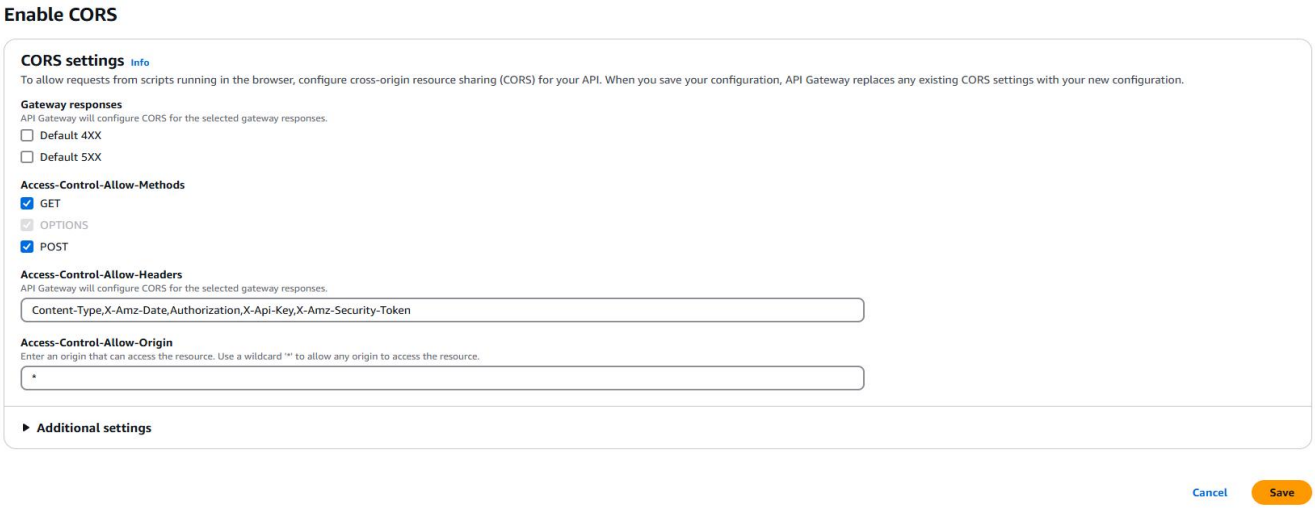
Cancel | Create method



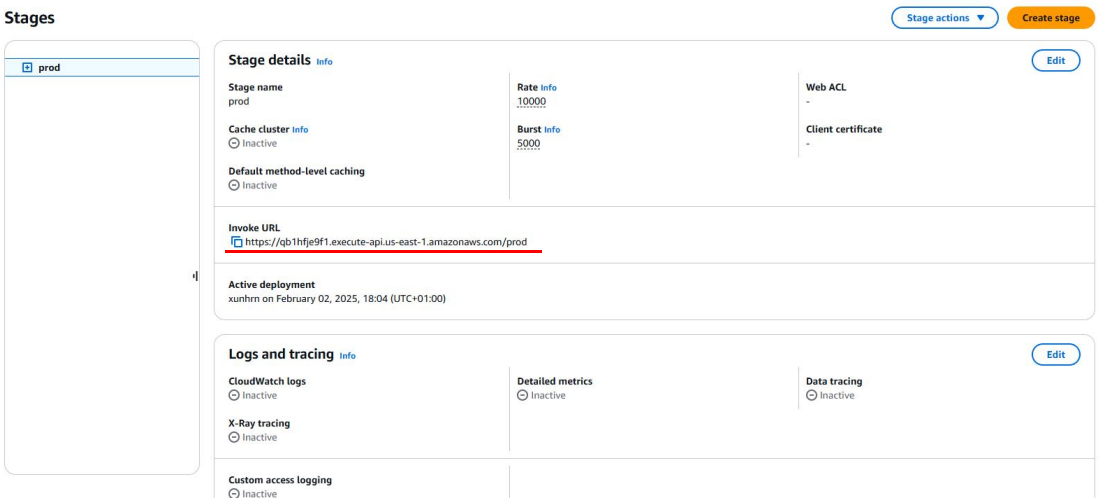
Then we deploy our API:



!! Also enable CORS:



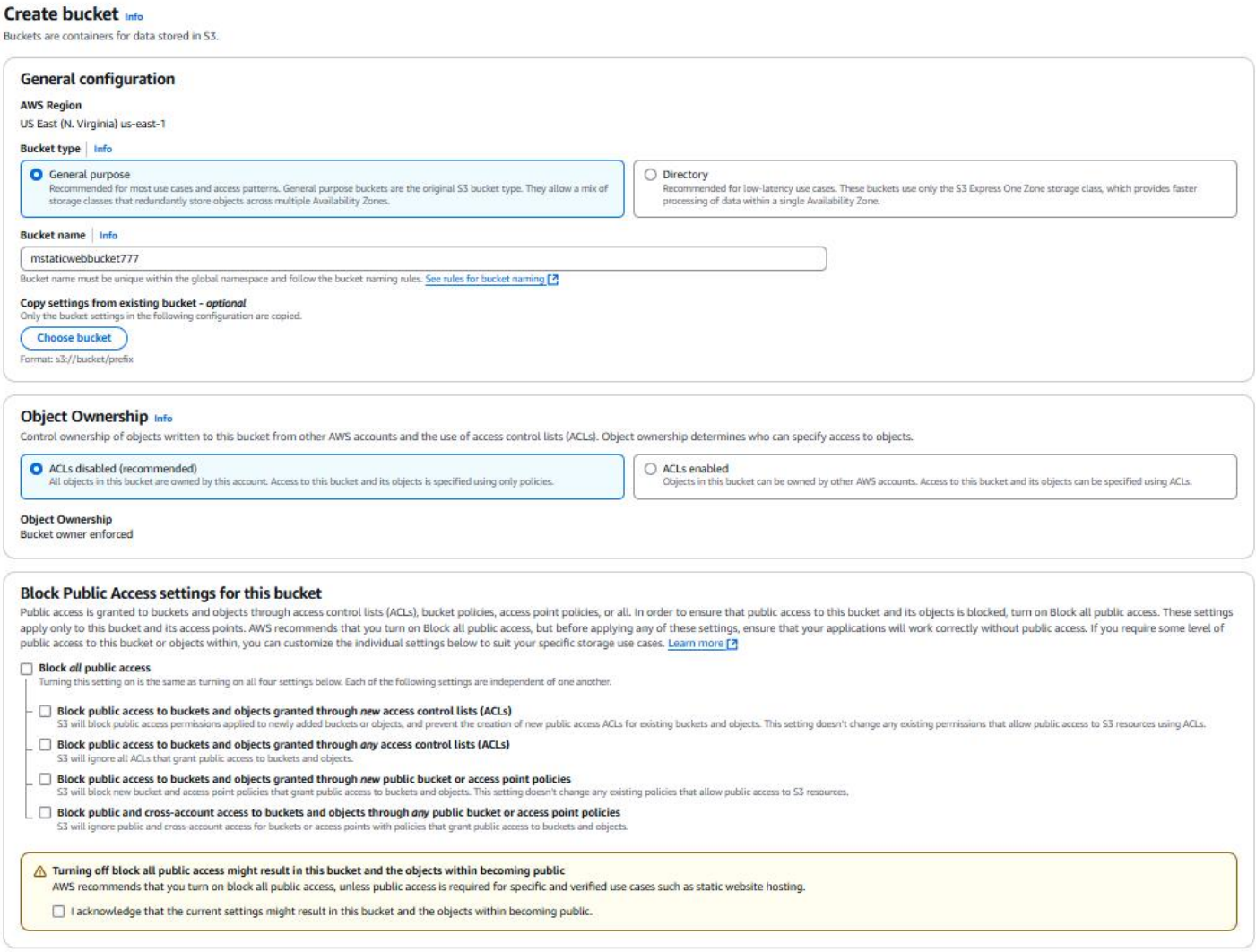
After this we should get an invoke URL



We'll paste in the variable 'apiEndpoint' in the **index.html** file before using S3.

Step3: Creating an S3 static web hosting:

We create an S3 bucket and enable static web hosting like it is shown below:



Bucket Versioning

Versioning is a means of keeping multiple variants of an object in the same bucket. You can use versioning to preserve, retrieve, and restore every version of every object stored in your Amazon S3 bucket. With versioning, you can easily recover from both unintended user actions and application failures. Learn more

Bucket Versioning

- (●) Disable
- ( ) Enable

Tags - *optional* (0)

You can use bucket tags to track storage costs and organize buckets. Learn more

No tags associated with this bucket.

Add tag

Default encryption Info

Server-side encryption is automatically applied to new objects stored in this bucket.

Encryption type | Info

- (●) Server-side encryption with Amazon S3 managed keys (SSE-S3)
- ( ) Server-side encryption with AWS Key Management Service keys (SSE-KMS)
- ( ) Dual-layer server-side encryption with AWS Key Management Service keys (DSSE-KMS)
  Secure your objects with two separate layers of encryption. For details on pricing, see **DSSE-KMS pricing** on the **Storage** tab of the Amazon S3 pricing page.

Bucket Key

Using an S3 Bucket Key for SSE-KMS reduces encryption costs by lowering calls to AWS KMS. S3 Bucket Keys aren't supported for DSSE-KMS. Learn more

- ( ) Disable
- (●) Enable

▸ **Advanced settings**

ⓘ After creating the bucket, you can upload files and folders to the bucket, and configure additional bucket settings.

Cancel | Create bucket

We enable static web hosting:

Edit static website hosting Info

Static website hosting

Use this bucket to host a website or redirect requests. Learn more

Static website hosting

- ( ) Disable
- (●) Enable

Hosting type

- (●) Host a static website
  Use the bucket endpoint as the web address. Learn more
- ( ) Redirect requests for an object
  Redirect requests to another bucket or domain. Learn more

ⓘ For your customers to access content at the website endpoint, you must make all your content publicly readable. To do so, you can edit the S3 Block Public Access settings for the bucket. For more information, see Using Amazon S3 Block Public Access

Index document

Specify the home or default page of the website.

index.html

Error document - *optional*

This is returned when an error occurs.

error.html

Redirection rules – *optional*

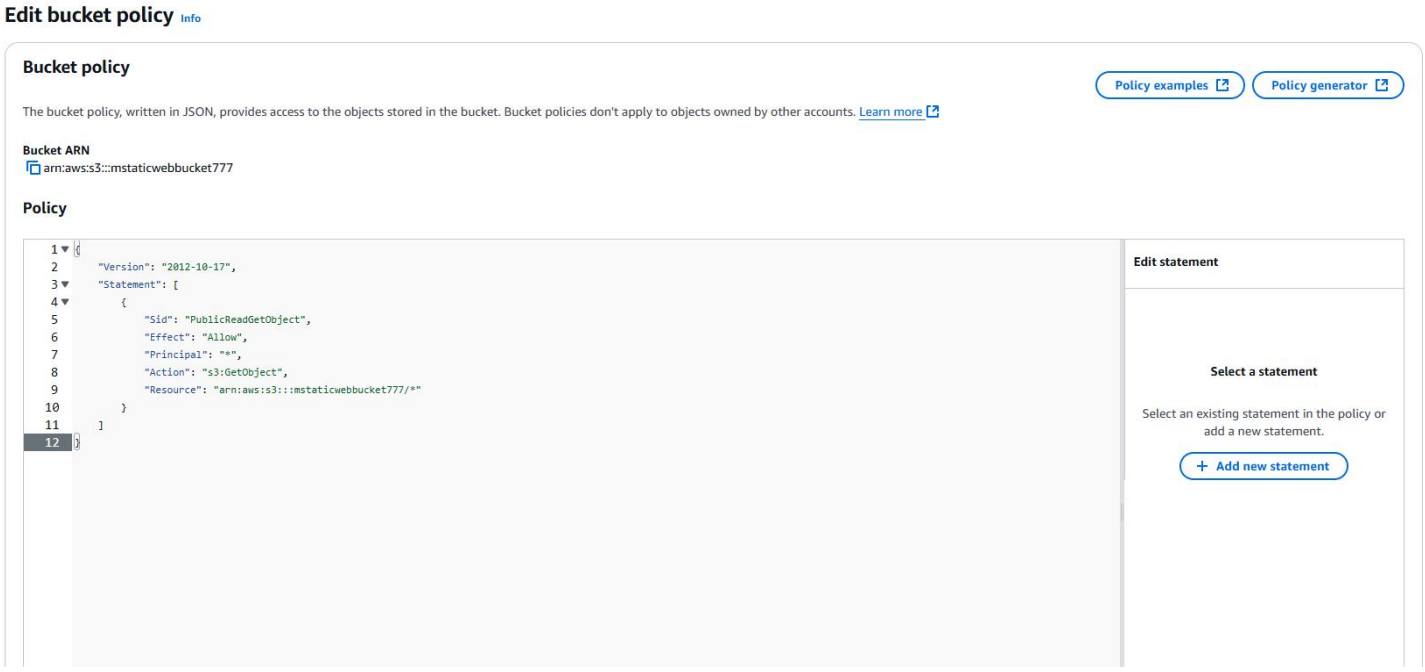
Redirection rules, written in JSON, automatically redirect webpage requests for specific content. Learn more

1

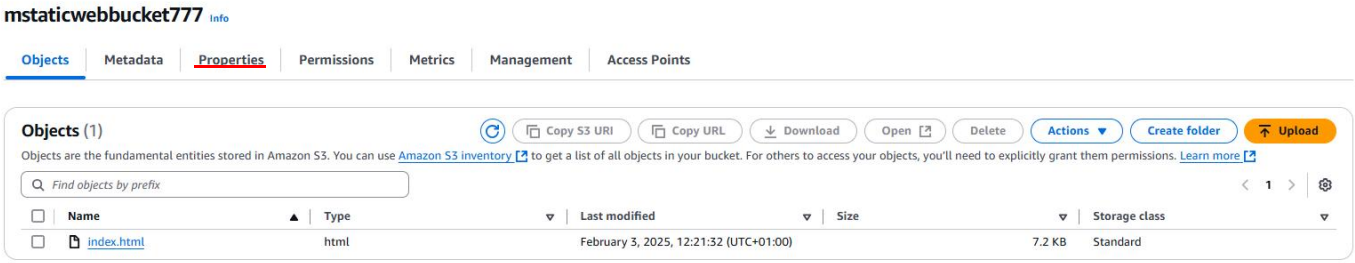
JSON Ln 1, Col 1 | Errors: 0 | Warnings: 0

Cancel | Save changes

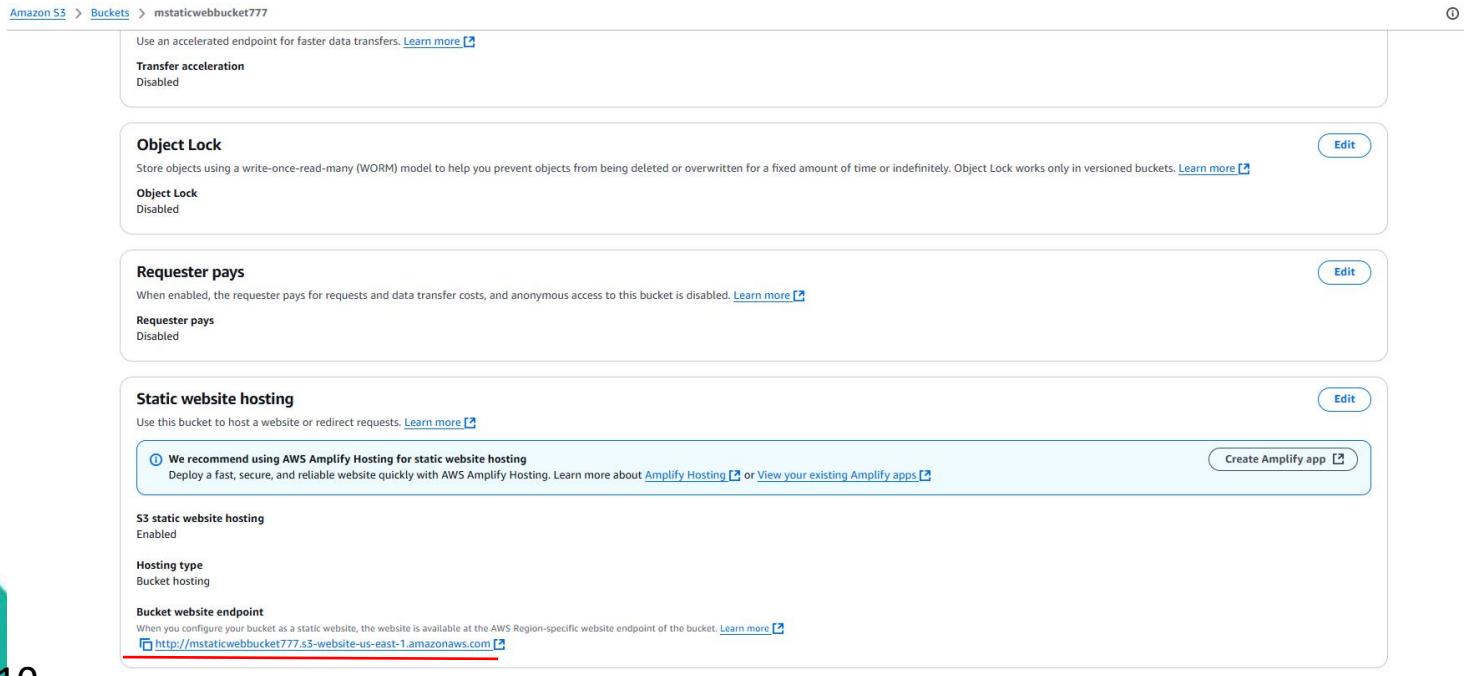
and we give it the following policy:



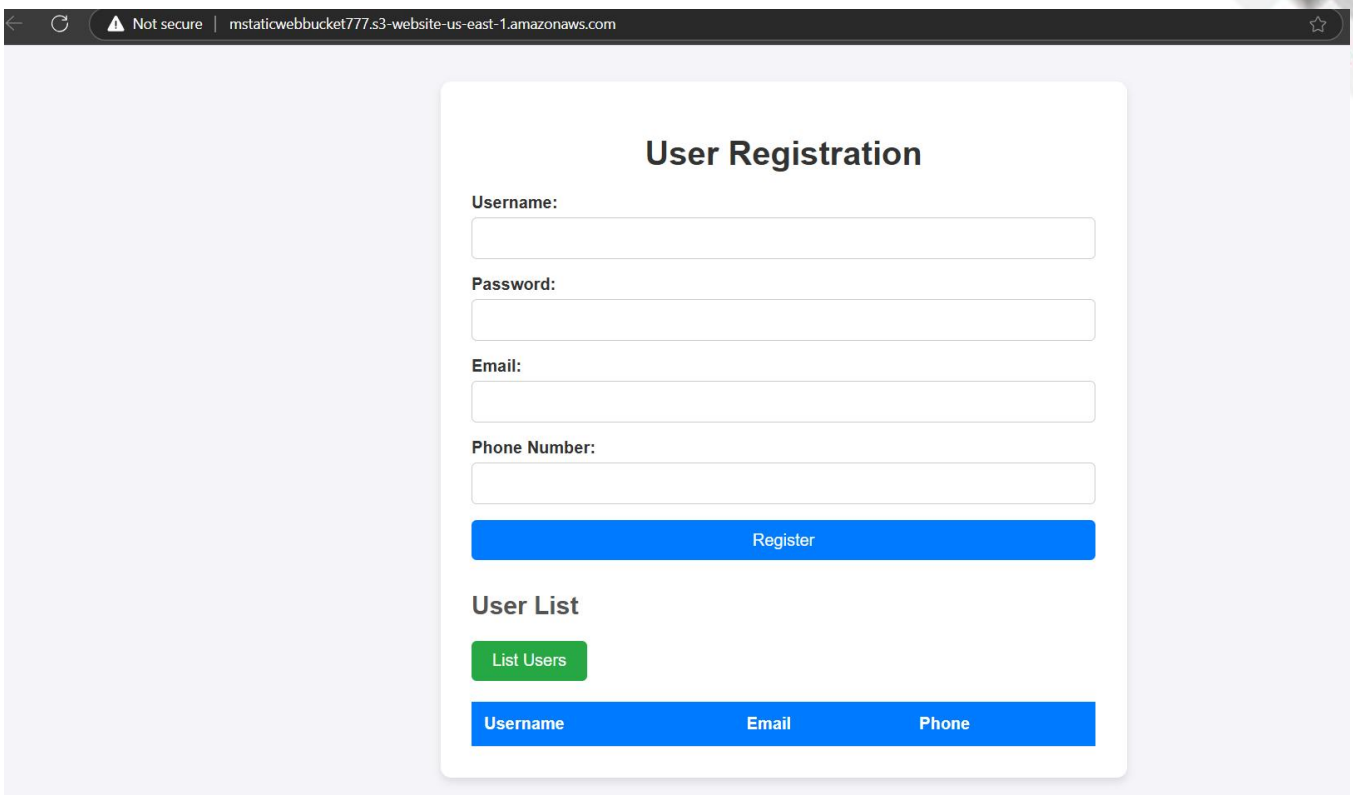
After uploading the **index.html** file, we go now to get the link in the <u>properties</u> section:



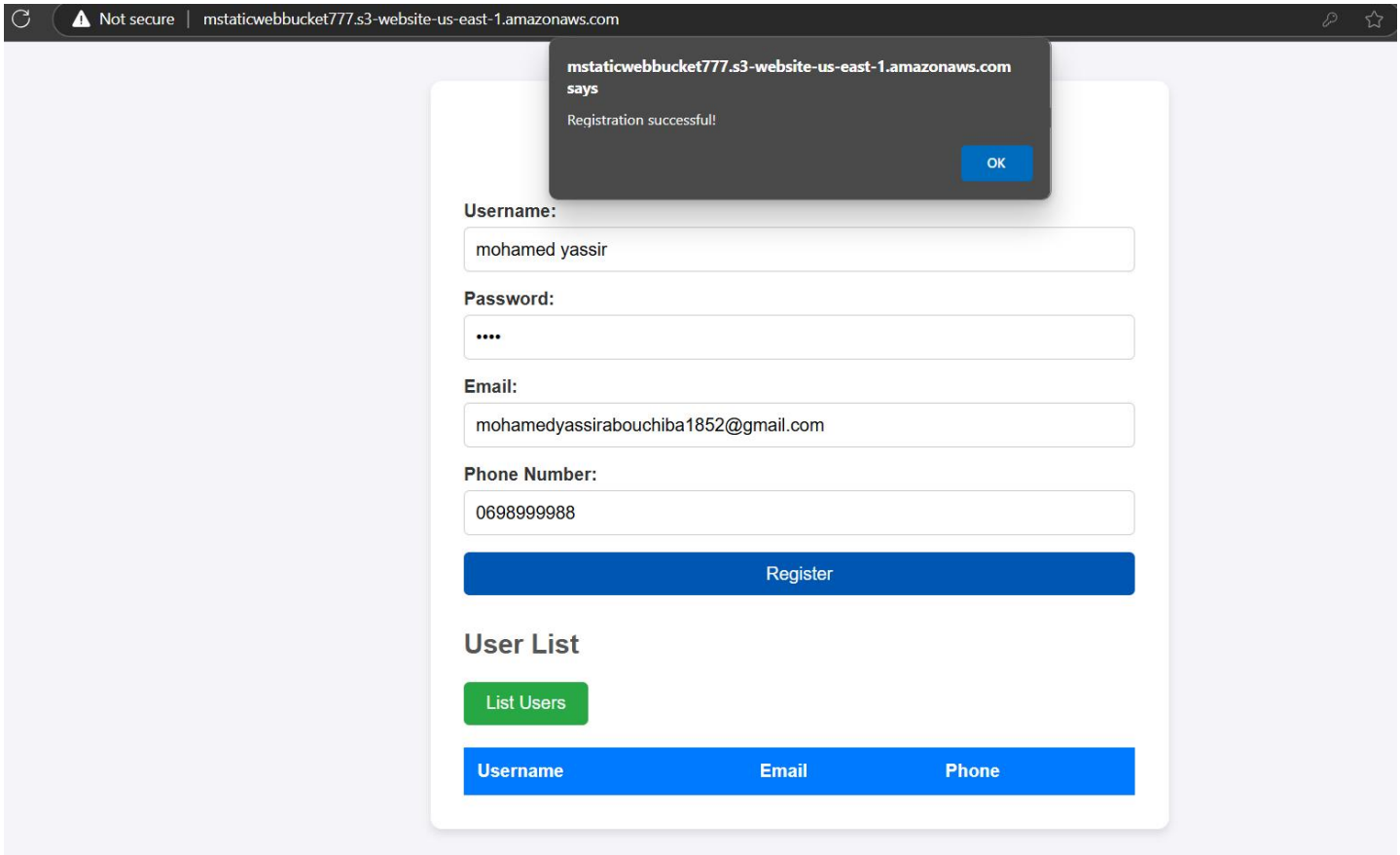
The link is located on this section, we click on it



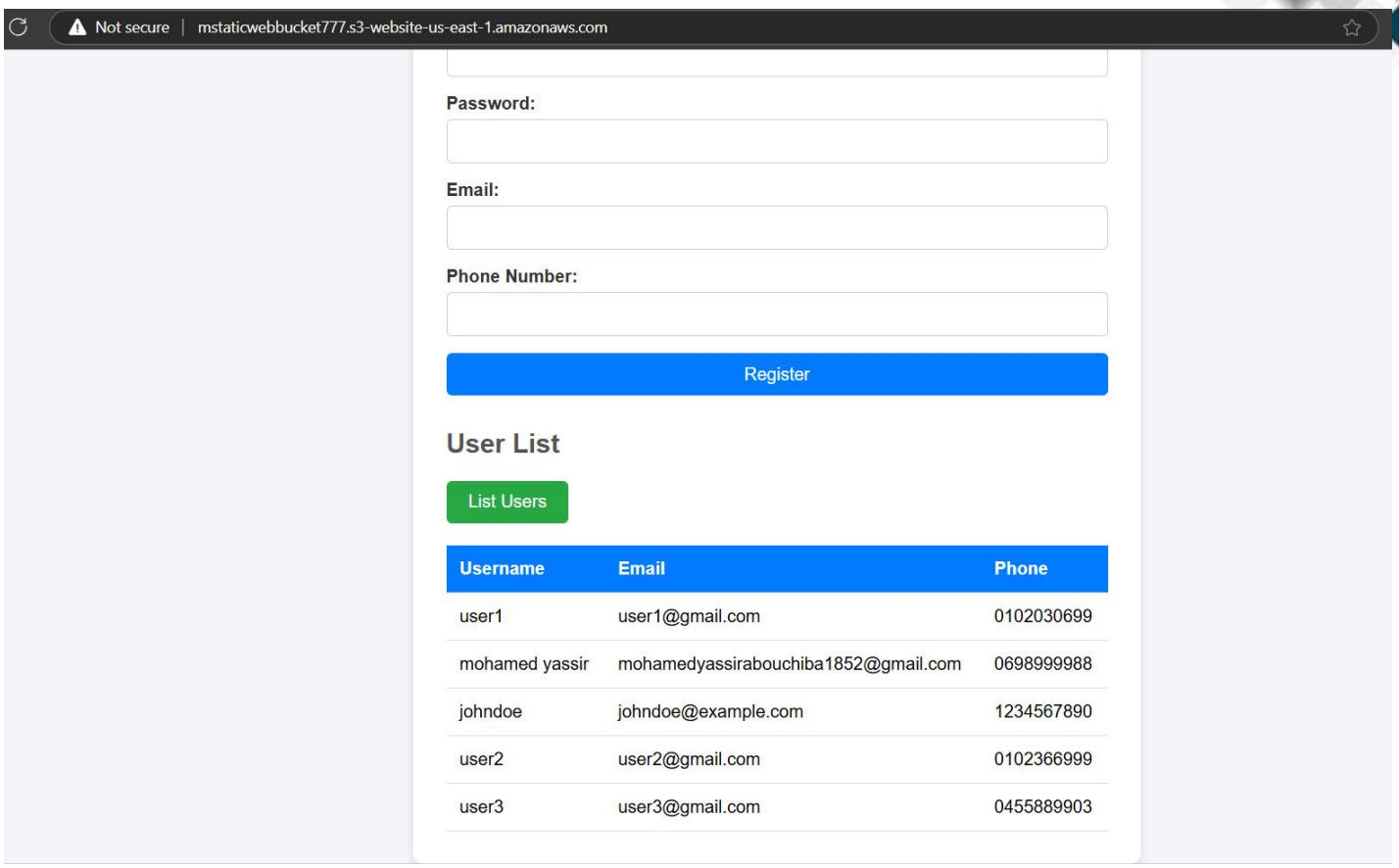
Now, we should get this:



If we try to register a user, it shows the following:



If we try now to list Users, it shows the following:



This means that the operation of reading and writing to the DynamoDB was successful.

Conclusion

This project has been an insightful journey into building a fully serverless web application using AWS services. By leveraging AWS Lambda, API Gateway, DynamoDB, and S3, I successfully implemented a scalable and cost-efficient user registration and management system within the AWS Free Tier.

Challenges Faced & Lessons Learned:

1) CORS Configuration in API Gateway & Lambda

One of the major challenges was handling CORS (Cross-Origin Resource Sharing) errors when making API requests from the frontend to API Gateway.
I learned how to properly configure CORS headers in API Gateway and Lambda responses to allow secure communication between the frontend and backend.

2) Setting Up API Gateway & Lambda Integration

Initially, API Gateway didn't route requests correctly to the appropriate Lambda functions.
I learned how to properly configure method integrations in API Gateway, ensuring that GET and POST requests were correctly processed by their respective Lambda functions.

3) S3 Bucket Permissions & Web Hosting

Hosting the frontend on Amazon S3 required configuring the right public access permissions and an S3 Bucket Policy to allow users to access the web application.
I learned how to correctly set up S3 static website hosting, adjust bucket policies, and ensure that the "Block Public Access" settings didn't interfere with serving the frontend.

Final Takeaways:

Through this project, I gained hands-on experience in serverless web development and learned how AWS services work together to create a fully functional application. Some key skills I developed include:

- ✅ Designing and deploying REST APIs with API Gateway & Lambda
- ✅ Configuring CORS policies for secure cross-origin requests
- ✅ Managing DynamoDB for efficient data storage and retrieval
- ✅ Setting up an S3-hosted static website with proper security policies

This project has deepened my understanding of serverless architecture and its advantages in terms of scalability, cost-efficiency, and ease of deployment. Moving forward, I am excited to apply these skills to build even more advanced cloud-based solutions.