AWS Project : Image recognition 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/Facial_Recognition_App

Introduction

Project Description

This project demonstrates the development of a fully serverless face recognition application using Amazon Web Services (AWS). The system is designed to provide a secure and efficient way for administrators to register employees and for employees to authenticate themselves using facial recognition technology.

Administrators can register employees by submitting their name and an image, which is then processed and stored using AWS Rekognition for facial recognition. The employee's facial features are indexed in a Rekognition Collection to enable future authentication. Employees can log in by uploading an image, and the system verifies their identity by comparing the uploaded image against the stored employee database. If a match is found, the employee is successfully authenticated.

Through this project, I gained hands-on experience in serverless architecture, facial recognition systems, cloud security, and AWS service integration, making it a valuable exploration of AI-powered authentication solutions.

Key Features

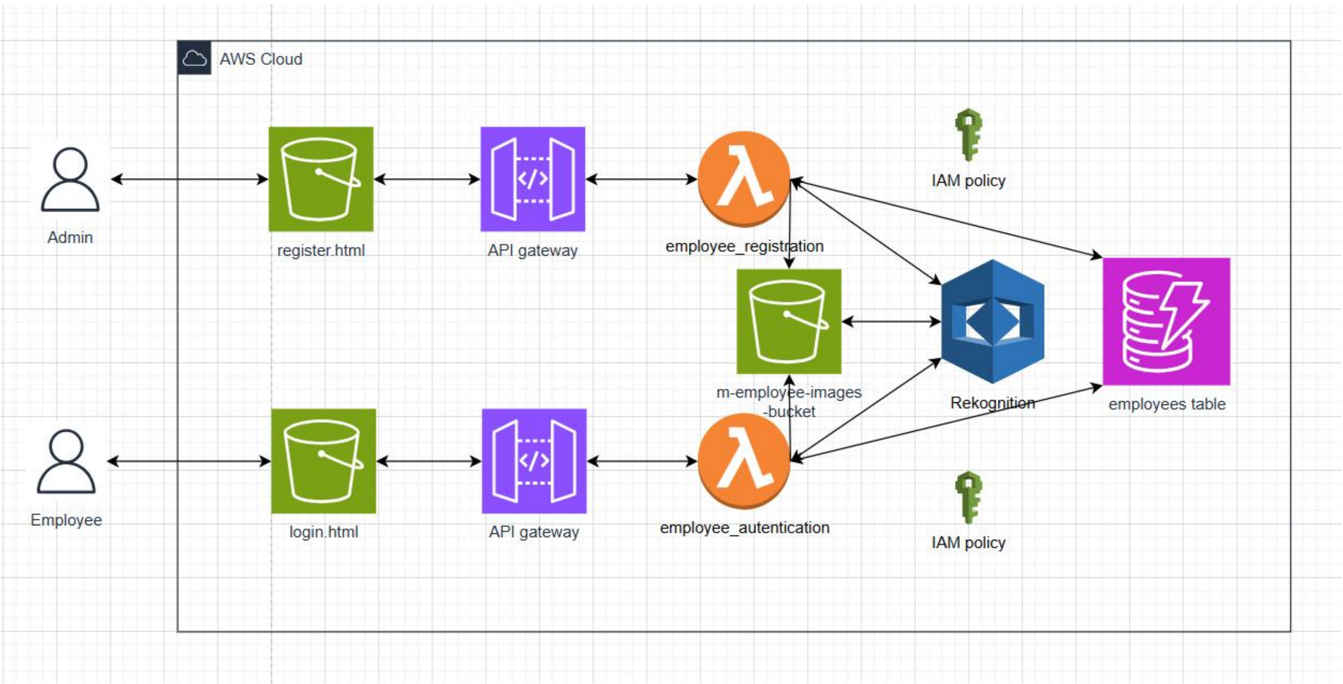
- Employee Registration: Admins can register employees by submitting their name and an image. The image is uploaded to an S3 bucket, and the face is indexed in a Rekognition collection. Employee details (name, ID, and image URL) are stored in a DynamoDB table. **Enhanced security** is ensured by denying registration if an employee's face is already registered, even if the image is different (e.g., a different pose or angle), preventing duplicate registrations.

- Employee Login: Employees can log in by uploading their image, and the system uses AWS Rekognition to match the face against the indexed collection. If a match is found, the employee is authenticated, and their details are displayed.

- Serverless Architecture: The application uses AWS Lambda for backend logic, API Gateway for RESTful APIs, and DynamoDB for data storage. It leverages Amazon S3 for image storage and Amazon Rekognition for facial recognition, ensuring scalability, cost-efficiency, and ease of deployment.

- Static Website Hosting: The frontend, built with HTML, CSS, and JavaScript, is hosted on Amazon S3 for cost-effective and scalable static website hosting.

AWS architecture diagram



- Employee registration workflow:

Admins submit an employee's name and image. The image is temporarily uploaded to S3, and Rekognition checks if the face is already indexed. If no match is found, the image is stored permanently in S3, employee details are saved in DynamoDB, and the face is indexed in Rekognition for future recognition.

- Employee login workflow:

Employees upload their image, which is temporarily stored in S3. Rekognition searches for the face in its collection. If a match is found, the employee's details are retrieved from DynamoDB and displayed. If no match is found, an error message is shown.

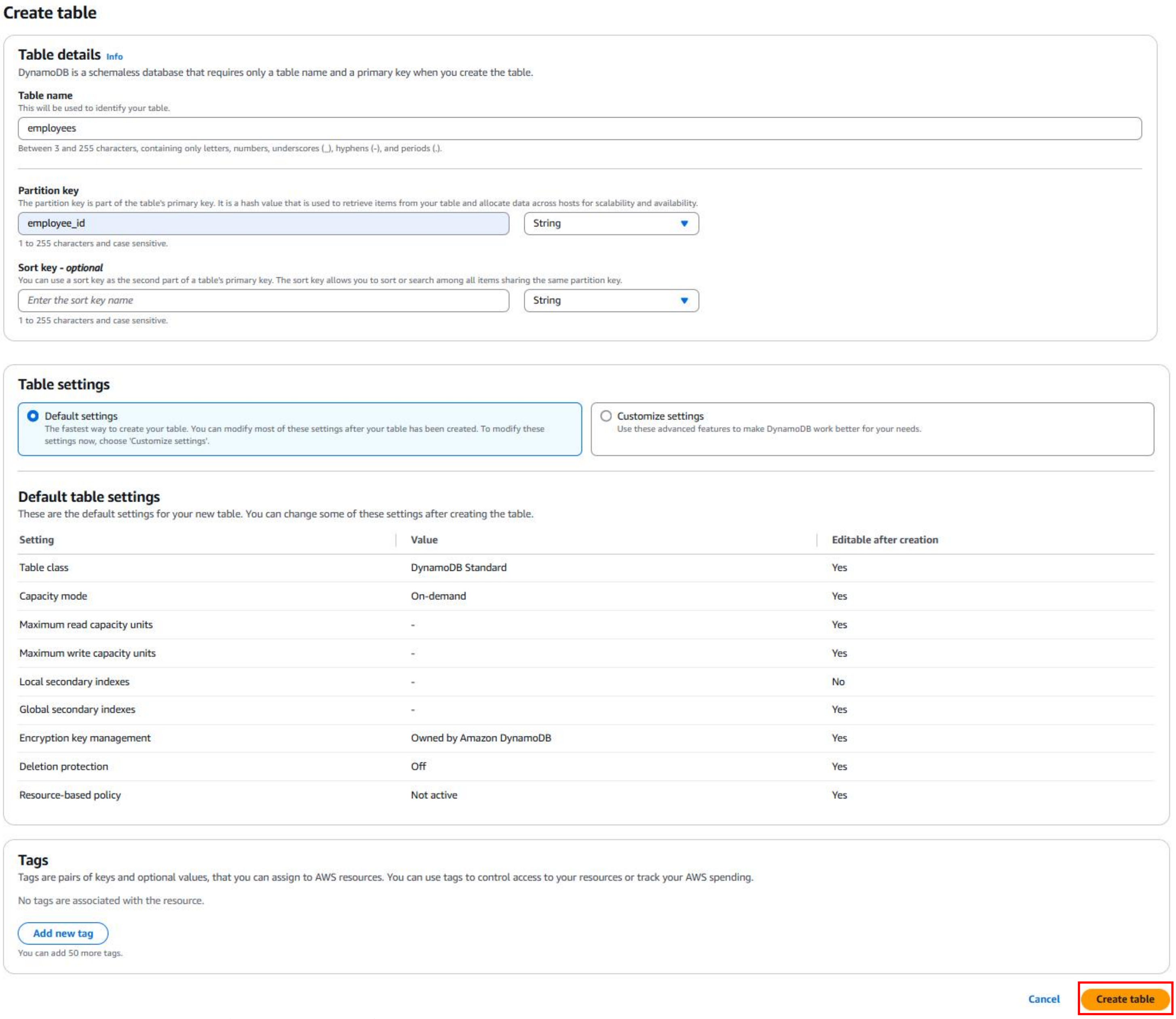
Step-by-Step implementation

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

Step1: Create a DynamoDB table and the S3 bucket:

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.
employees
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.
employee_id — 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.

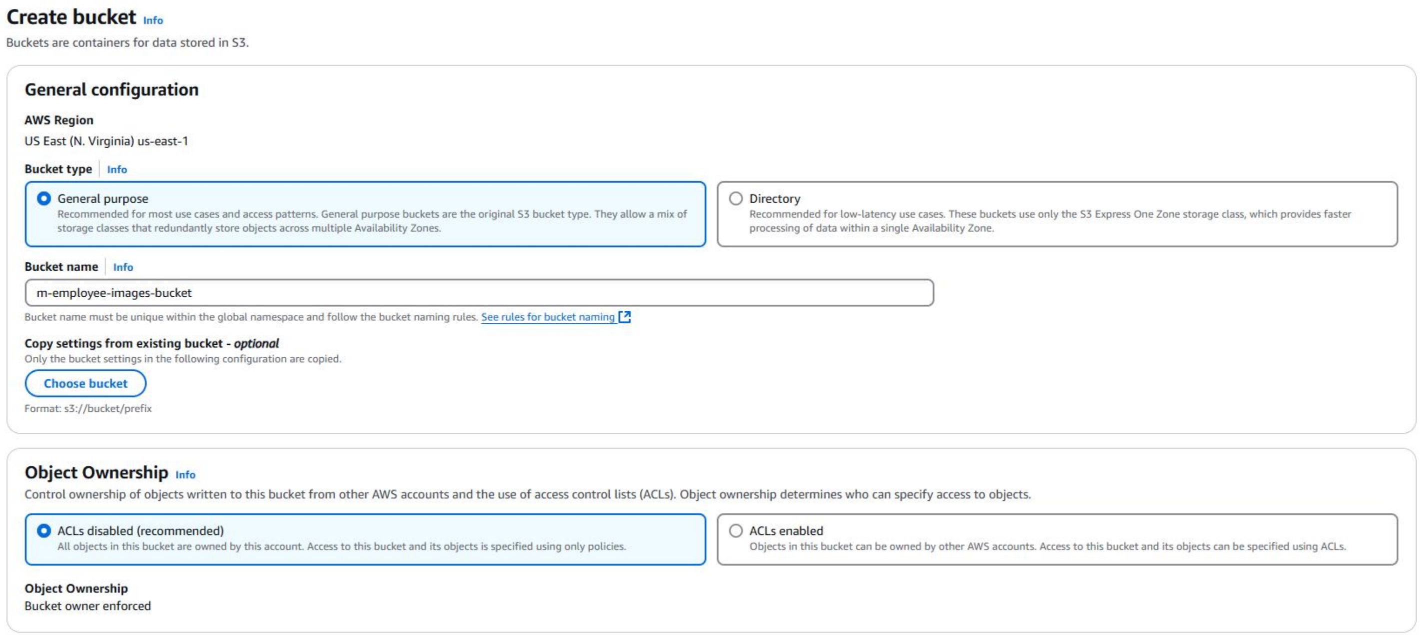
Setting	Value	Editable after creation
Table class	DynamoDB Standard	Yes
Capacity mode	On-demand	Yes
Maximum read capacity units	-	Yes
Maximum write capacity units	-	Yes
Local secondary indexes	-	No
Global secondary indexes	-	Yes
Encryption key management	Owned by Amazon DynamoDB	Yes
Deletion protection	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.

You can add 50 more tags.

We click on '*Create table*'

We go to S3 to create our bucket, named ‘m-employee-images-bucket’



Create bucket Info

Buckets are containers for data stored in S3.

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
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

m-employee-images-bucket

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.

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.

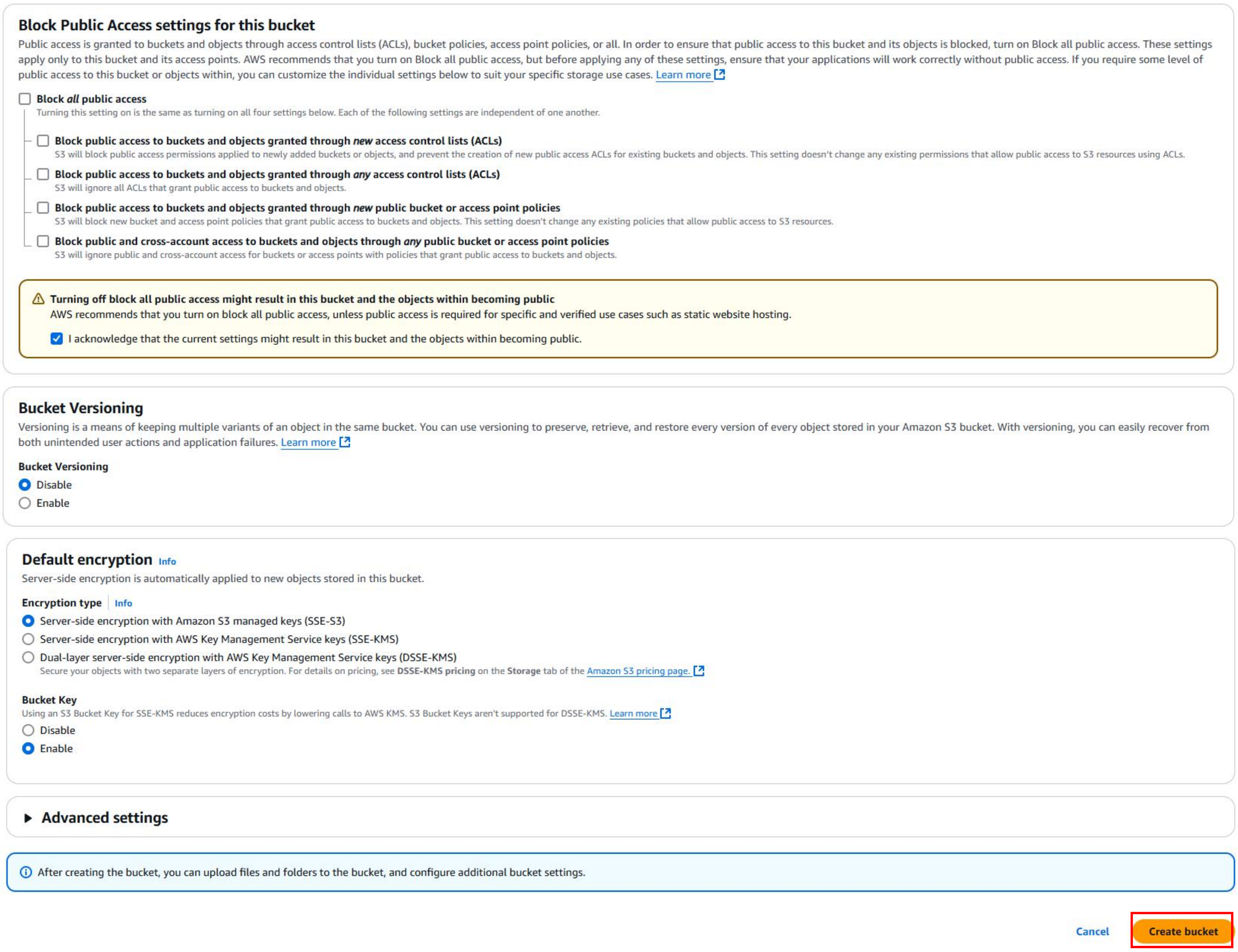
● ACLs disabled (recommended)
All objects in this bucket are owned by this account. Access to this bucket and its objects is specified using only policies.

○ 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

We need also to **disable** ‘Block public access to this bucket’ in order to let our lambda functions and Rekognition access it



Block Public Access settings for this bucket

Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that public access to this bucket and its objects is blocked, turn on Block all public access. These settings apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying any of these settings, ensure that your applications will work correctly without public access. If you require some level of public access to this bucket or objects within, you can customize the individual settings below to suit your specific storage use cases. Learn more

□ **Block *all* public access**
Turning this setting on is the same as turning on all four settings below. Each of the following settings are independent of one another.

- □ **Block public access to buckets and objects granted through *new* access control lists (ACLs)**
  S3 will block public access permissions applied to newly added buckets or objects, and prevent the creation of new public access ACLs for existing buckets and objects. This setting doesn't change any existing permissions that allow public access to S3 resources using ACLs.
- □ **Block public access to buckets and objects granted through *any* access control lists (ACLs)**
  S3 will ignore all ACLs that grant public access to buckets and objects.
- □ **Block public access to buckets and objects granted through *new* public bucket or access point policies**
  S3 will block new bucket and access point policies that grant public access to buckets and objects. This setting doesn't change any existing policies that allow public access to S3 resources.
- □ **Block public and cross-account access to buckets and objects through *any* public bucket or access point policies**
  S3 will ignore public and cross-account access for buckets or access points with policies that grant public access to buckets and objects.

⚠ **Turning off block all public access might result in this bucket and the objects within becoming public**
AWS recommends that you turn on block all public access, unless public access is required for specific and verified use cases such as static website hosting.

☑ I acknowledge that the current settings might result in this bucket and the objects within becoming public.

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

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

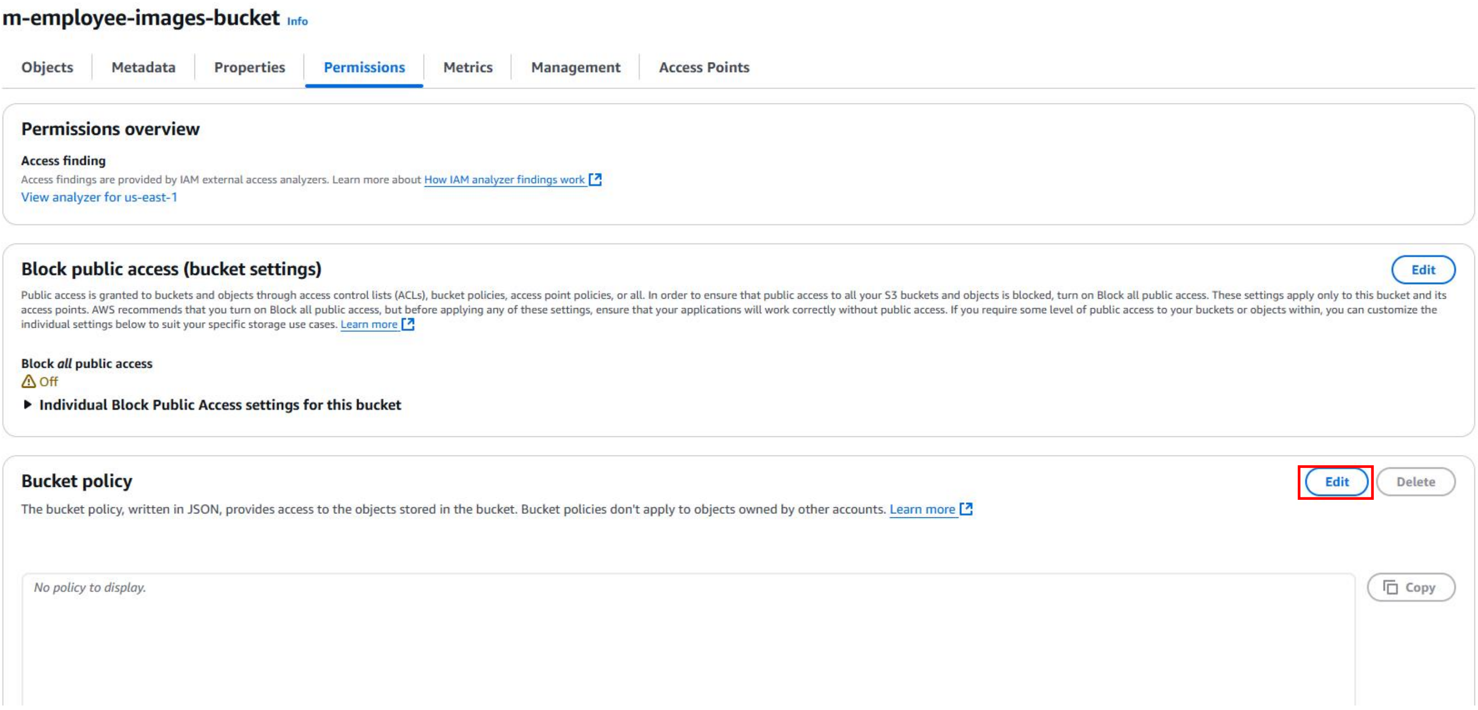
Using an S3 Bucket Key for SSE-KMS reduces encryption costs by lowering calls to AWS KMS. S3 Bucket Keys aren't supported for DSSE-KMS. Learn more

○ Disable
● Enable

▶ Advanced settings

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

After creating the bucket, we need to assign a policy to it in order to let rekognition access it, so we click on the bucket name, we go to the permissions tab and we click on ‘edit’ on the policy section



m-employee-images-bucket Info

Permissions overview

Access finding

Access findings are provided by IAM external access analyzers. Learn more about How IAM analyzer findings work

View analyzer for us-east-1

Block public access (bucket settings)

Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that public access to all your S3 buckets and objects is blocked, turn on Block all public access. These settings apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying any of these settings, ensure that your applications will work correctly without public access. If you require some level of public access to your buckets or objects within, you can customize the individual settings below to suit your specific storage use cases. Learn more

Block *all* public access

⚠ Off

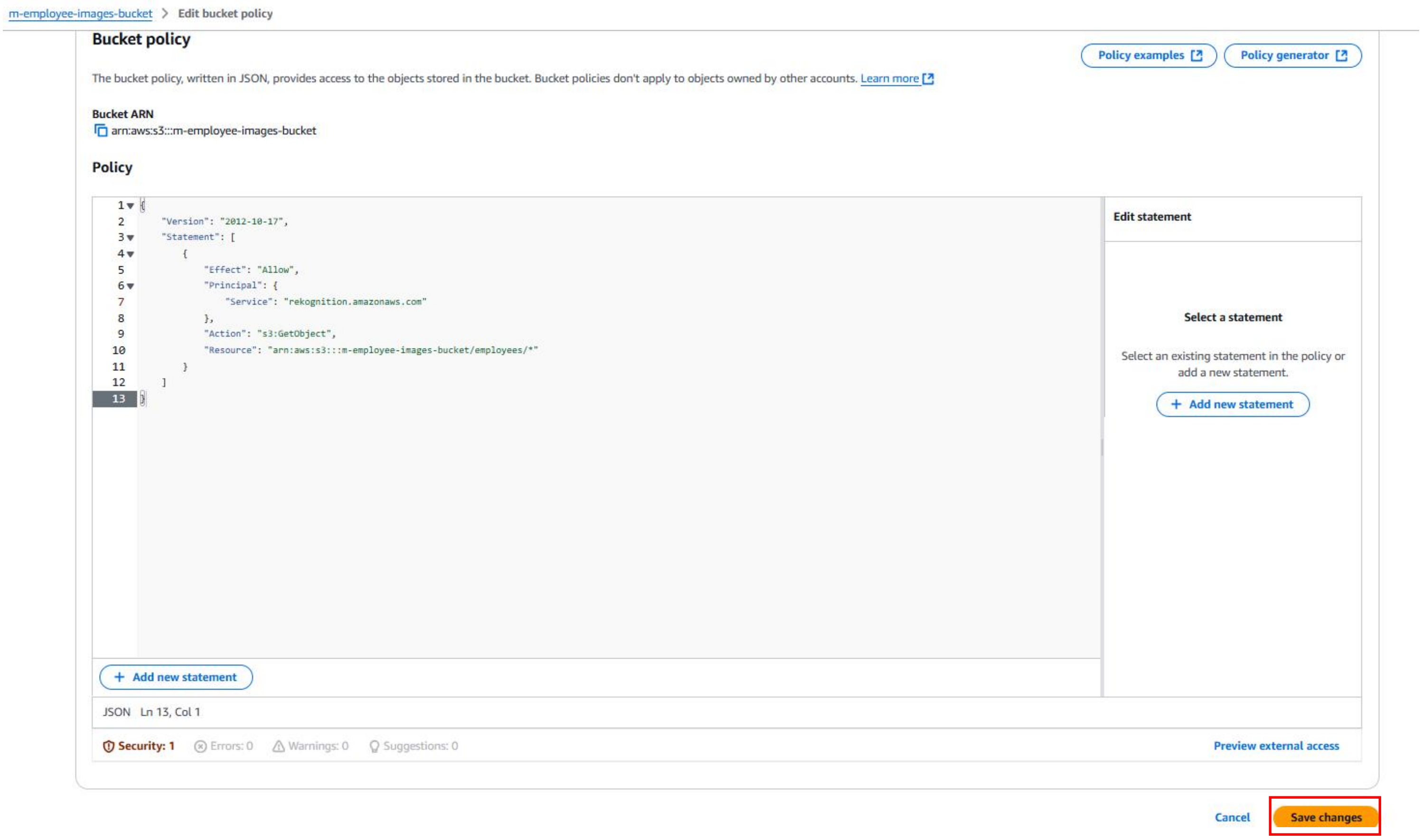
▶ **Individual Block Public Access settings for this bucket**

Bucket policy

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

No policy to display.

Then, we paste the policy (the policy is on the repository)



m-employee-images-bucket > Edit bucket policy

Bucket policy

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:::m-employee-images-bucket

Policy

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "rekognition.amazonaws.com"
            },
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::m-employee-images-bucket/employees/*"
        }
    ]
}
```

Edit statement

Select a statement

Select an existing statement in the policy or add a new statement.

JSON Ln 13, Col 1

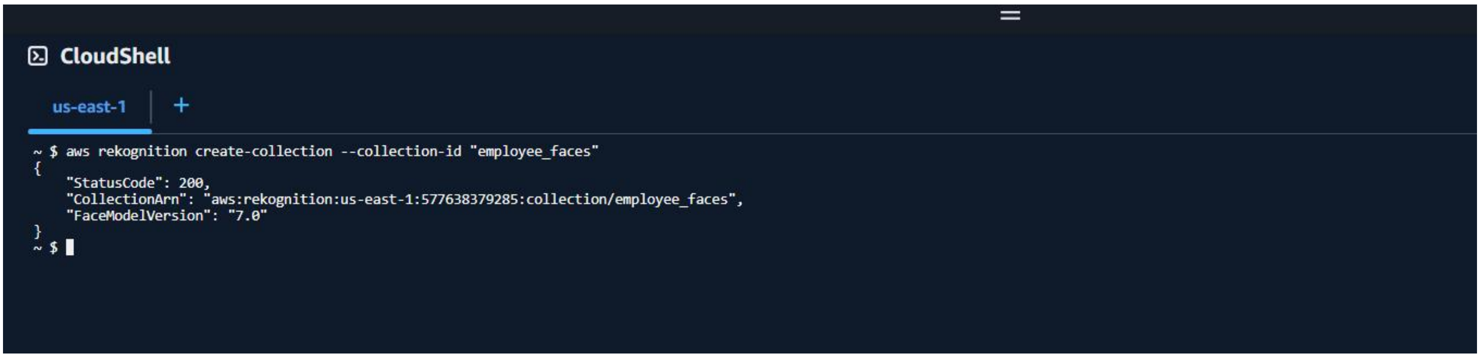
Security: 1 Errors: 0 Warnings: 0 Suggestions: 0

Step2: Create a rekognition collection, create the employee_regitration lambda function and trigger it with an API gateway:

To create a rekognition collection, we go to CloudShell, write the following command:

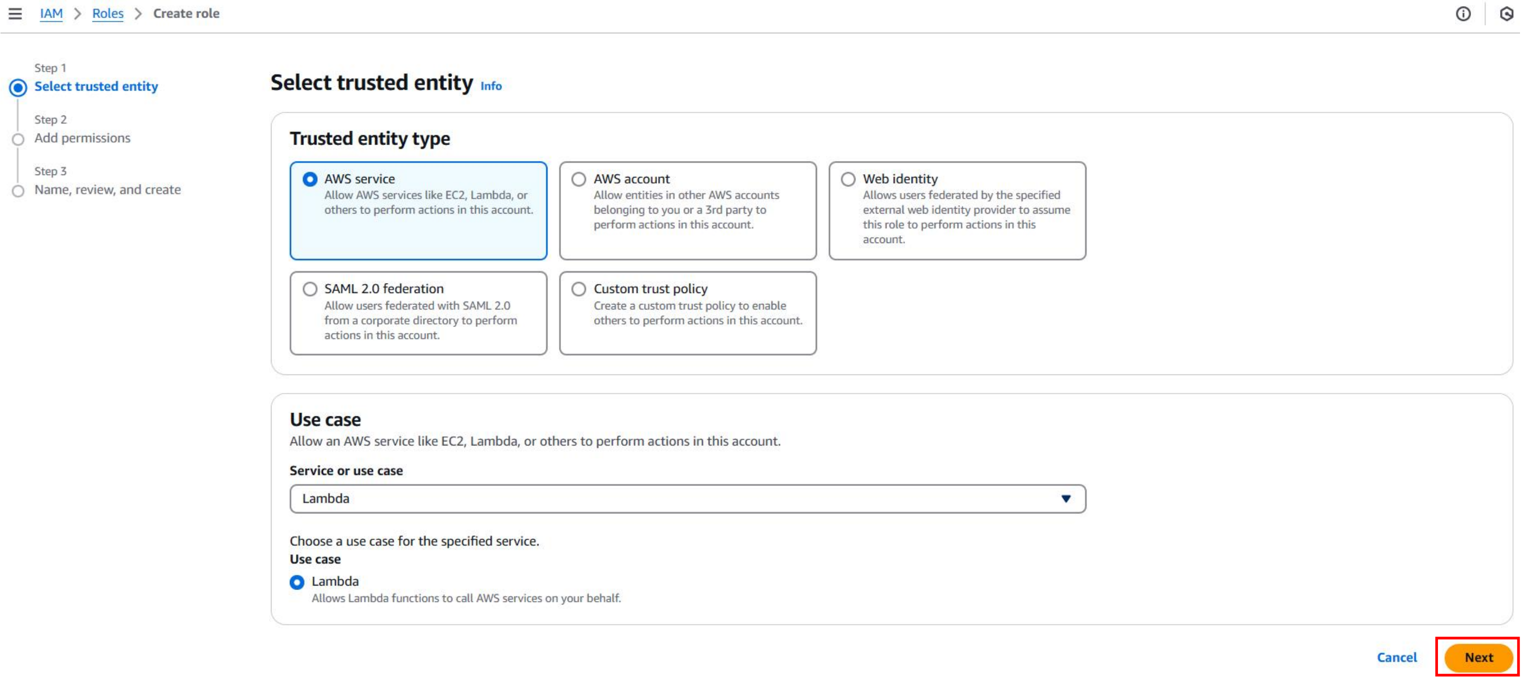
aws rekognition create-collection --collection-id "employee_faces"

We should get this output:



```
~ $ aws rekognition create-collection --collection-id "employee_faces"
{
    "StatusCode": 200,
    "CollectionArn": "aws:rekognition:us-east-1:577638379285:collection/employee_faces",
    "FaceModelVersion": "7.0"
}
~ $ 
```

Before creating the lambda functions, we should create a role because without it, lambda cannot access our resources, so we go to IAM>Roles>Create role, then we fill in the following informations like it is shown below:



Select trusted entity

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.

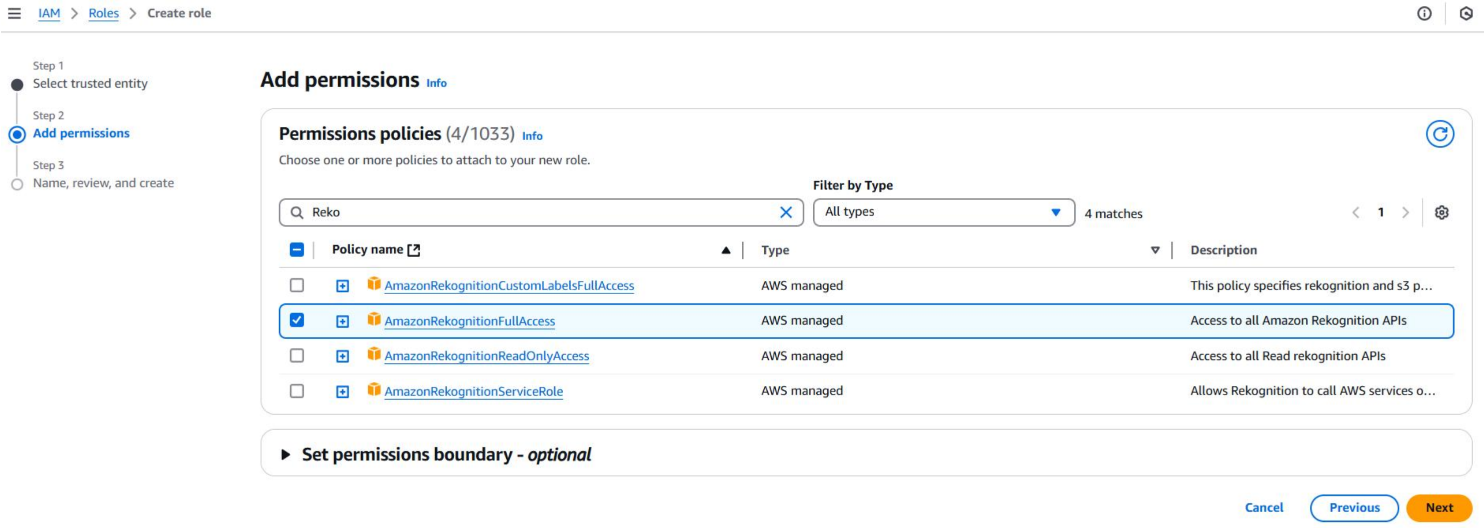
Service or use case: Lambda

Choose a use case for the specified service.

Use case

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

We add the following permissions: **AmazonS3FullAccess, AmazonDynamoDBFullAccess, CloudWatchFullAccess, AmazonRekognitionFullAccess** by searching their names and selecting them like it is shown for example for **AmazonRekognitionFullAccess**



Add permissions Info

Permissions policies (4/1033) Info

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

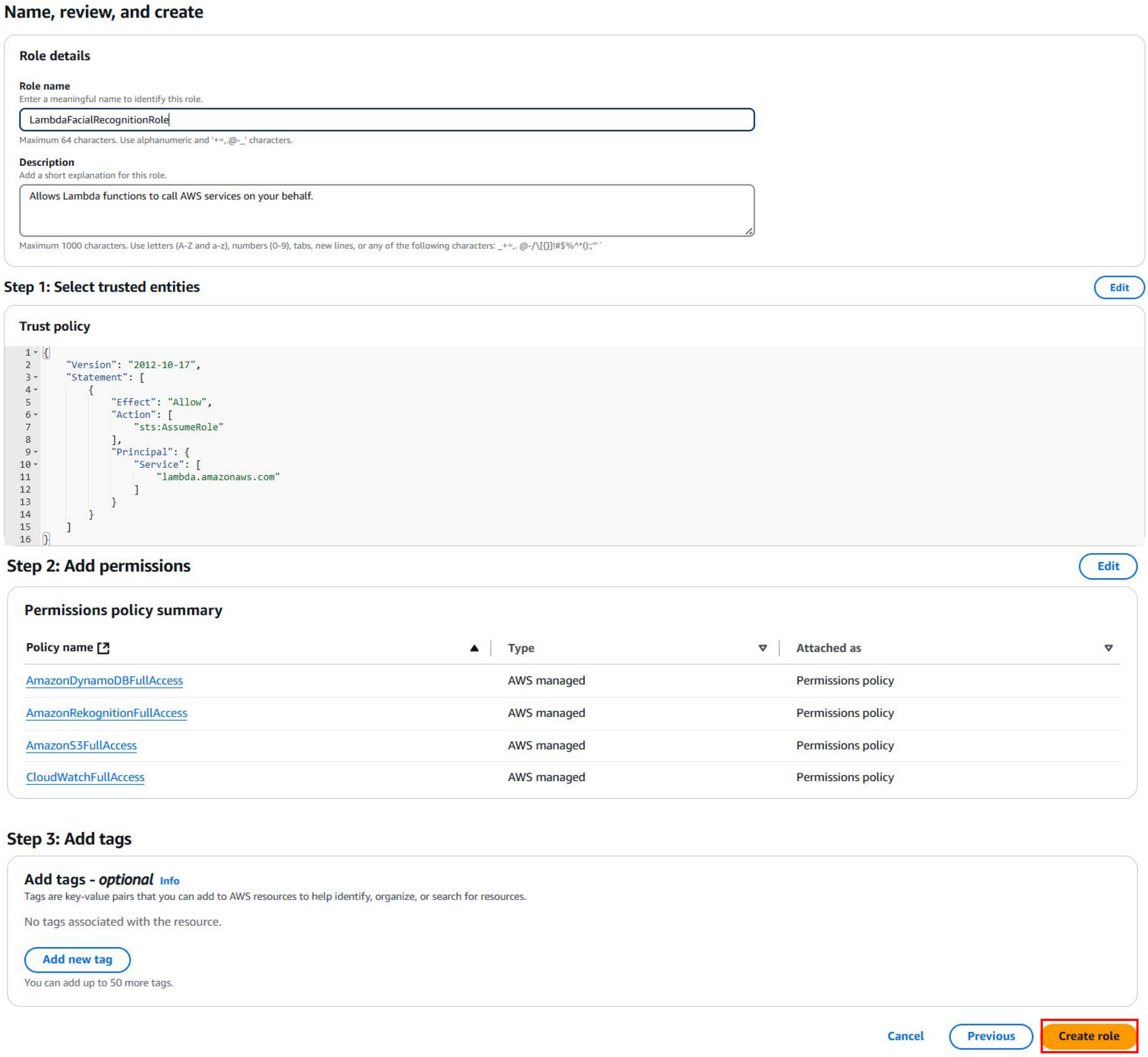
Filter by Type

Search: Reko — Filter by Type: All types — 4 matches

	Policy name	Type	Description
□	AmazonRekognitionCustomLabelsFullAccess	AWS managed	This policy specifies rekognition and s3 p…
☑	AmazonRekognitionFullAccess	AWS managed	Access to all Amazon Rekognition APIs
□	AmazonRekognitionReadOnlyAccess	AWS managed	Access to all Read rekognition APIs
□	AmazonRekognitionServiceRole	AWS managed	Allows Rekognition to call AWS services o…

▶ Set permissions boundary - *optional*

After adding all these policies, we click on ‘next’, we should get this:



Name, review, and create

Role details

Role name

Enter a meaningful name to identify this role.

LambdaFacialRecognitionRole

Maximum 64 characters. Use alphanumeric and '+=,.@-_' characters.

Description

Add a short explanation for this role.

Allows Lambda functions to call AWS services on your behalf.

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

Trust policy

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

Step 2: Add permissions

Permissions policy summary

Policy name	Type	Attached as
AmazonDynamoDBFullAccess	AWS managed	Permissions policy
AmazonRekognitionFullAccess	AWS managed	Permissions policy
AmazonS3FullAccess	AWS managed	Permissions policy
CloudWatchFullAccess	AWS managed	Permissions policy

Step 3: Add tags

Add tags - *optional* Info

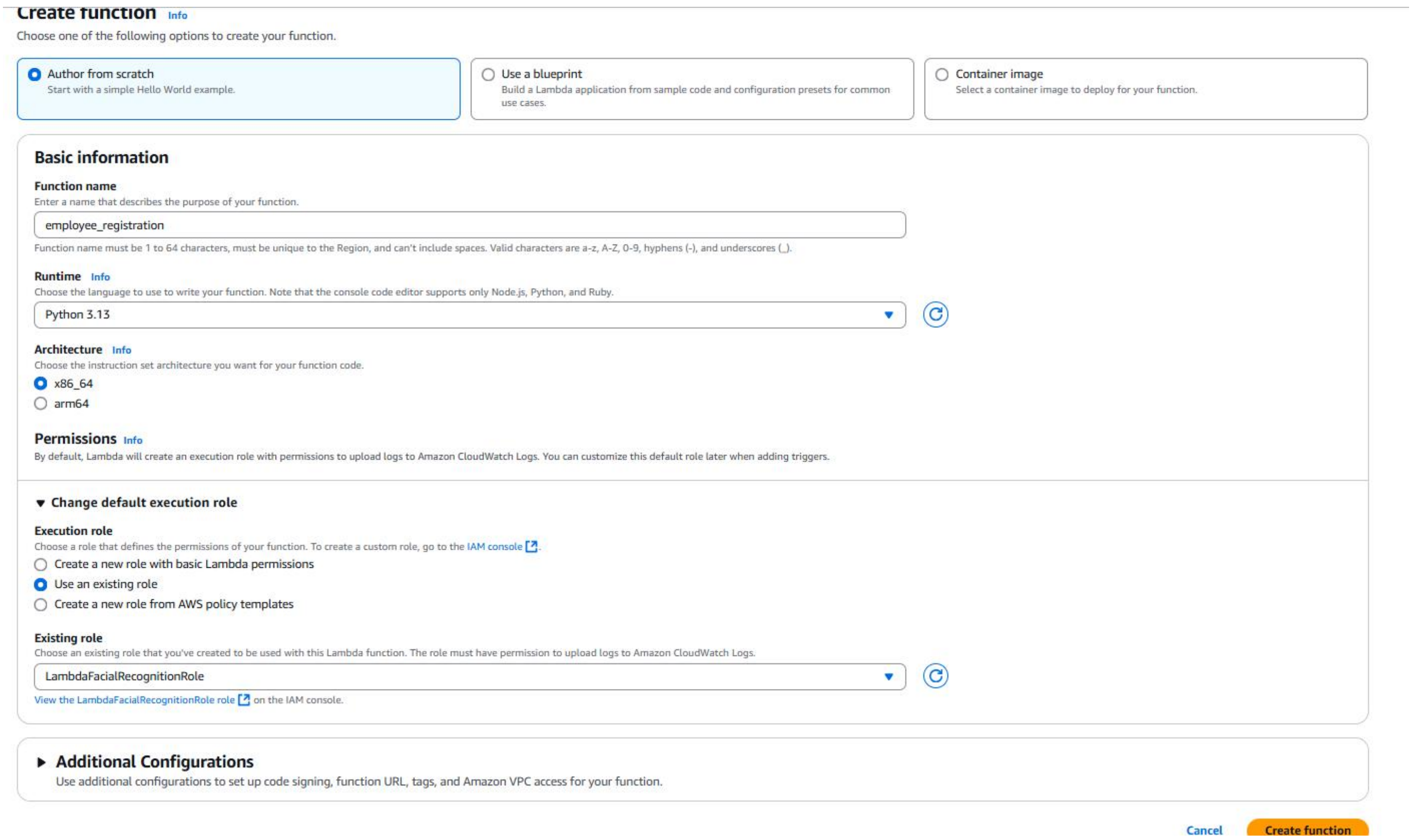
Tags are key-value pairs that you can add to AWS resources to help identify, organize, or search for resources.

No tags associated with the resource.

You can add up to 50 more tags.

We name it **LambdaFacialRecognitionRole** and we click on 'create role'

Next up, we go to lambda and we create a function named 'employee_registration' without forgetting to add the **LambdaFacialRecognitionRole** created before



Create function Info

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.

employee_registration

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

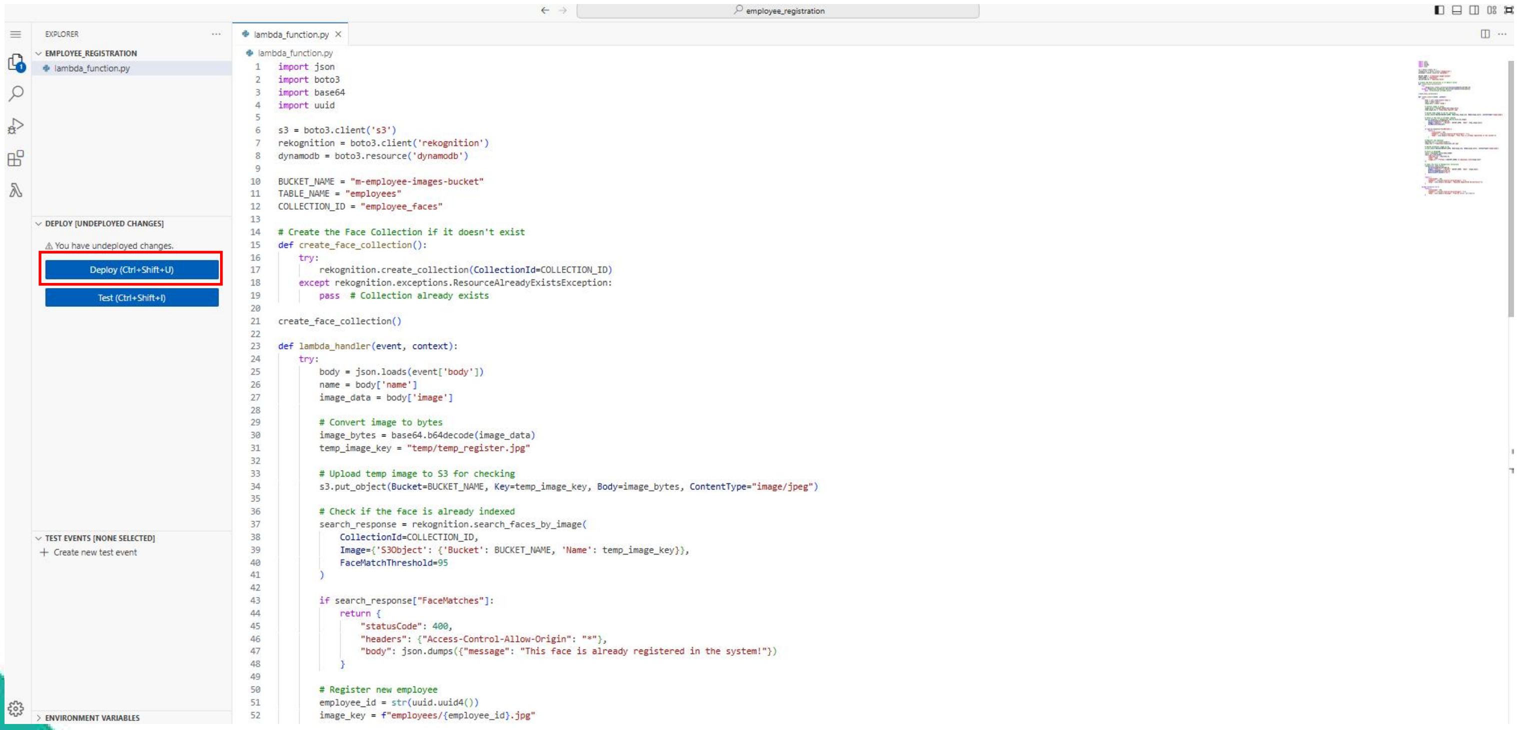
Existing role
Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.

LambdaFacialRecognitionRole

View the LambdaFacialRecognitionRole 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.

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):

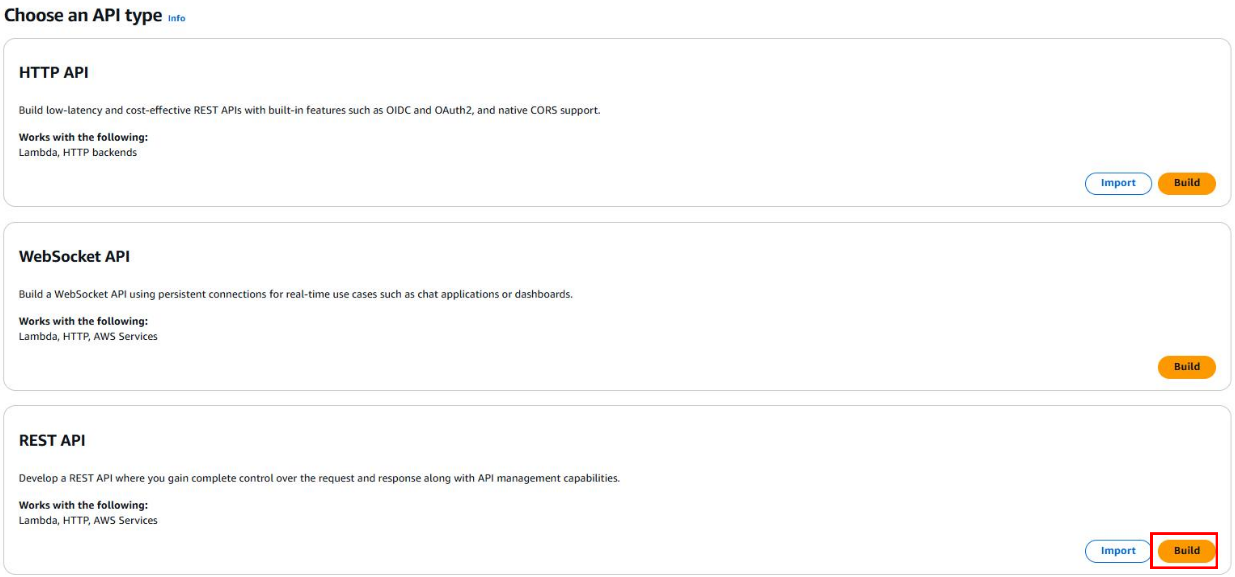


lambda_function.py

```python
import json
import boto3
import base64
import uuid

s3 = boto3.client('s3')
rekognition = boto3.client('rekognition')
dynamodb = boto3.resource('dynamodb')

BUCKET_NAME = "m-employee-images-bucket"
TABLE_NAME = "employees"
COLLECTION_ID = "employee_faces"

# Create the Face Collection if it doesn't exist
def create_face_collection():
    try:
        rekognition.create_collection(CollectionId=COLLECTION_ID)
    except rekognition.exceptions.ResourceAlreadyExistsException:
        pass  # Collection already exists

create_face_collection()

def lambda_handler(event, context):
    try:
        body = json.loads(event['body'])
        name = body['name']
        image_data = body['image']

        # Convert image to bytes
        image_bytes = base64.b64decode(image_data)
        temp_image_key = "temp/temp_register.jpg"

        # Upload temp image to S3 for checking
        s3.put_object(Bucket=BUCKET_NAME, Key=temp_image_key, Body=image_bytes, ContentType="image/jpeg")

        # Check if the face is already indexed
        search_response = rekognition.search_faces_by_image(
            CollectionId=COLLECTION_ID,
            Image={'S3Object': {'Bucket': BUCKET_NAME, 'Name': temp_image_key}},
            FaceMatchThreshold=95
        )

        if search_response["FaceMatches"]:
            return {
                "statusCode": 400,
                "headers": {"Access-Control-Allow-Origin": "*"},
                "body": json.dumps({"message": "This face is already registered in the system!"})
            }

        # Register new employee
        employee_id = str(uuid.uuid4())
        image_key = f"employees/{employee_id}.jpg"
```

We click on ‘Deploy’.

We go now to create a REST API on the API gateway



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

WebSocket API

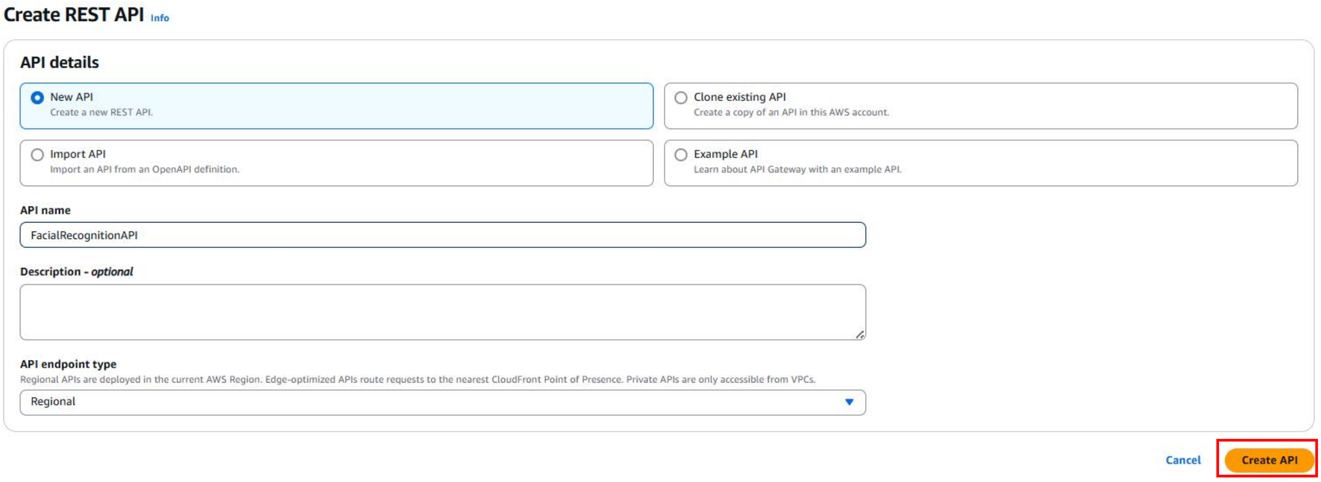
Build a WebSocket API using persistent connections for real-time use cases such as chat applications or dashboards.

Works with the following:
Lambda, HTTP, AWS Services

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



Create REST API Info

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.

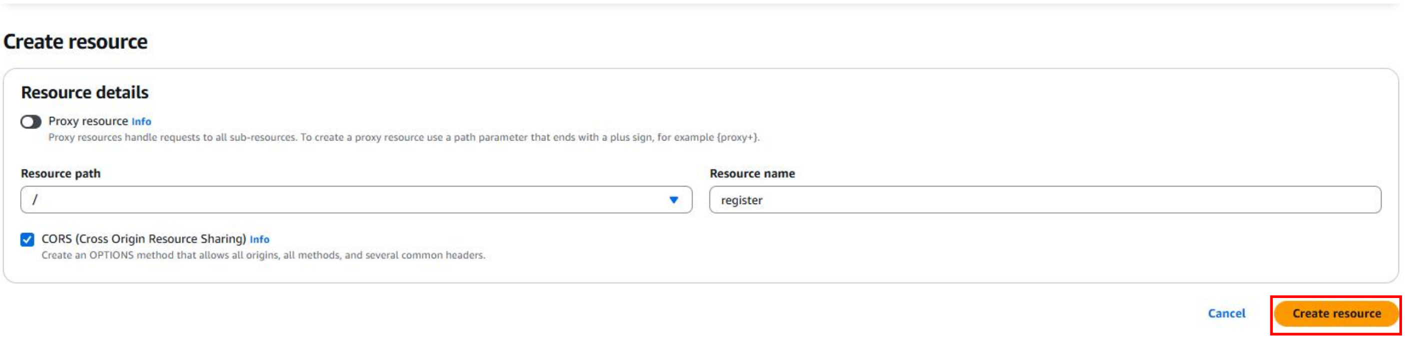
○ Example API
Learn about API Gateway with an example API.

API name
FacialRecognitionAPI

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.
Regional

We create a new resource named ‘register’



Create resource

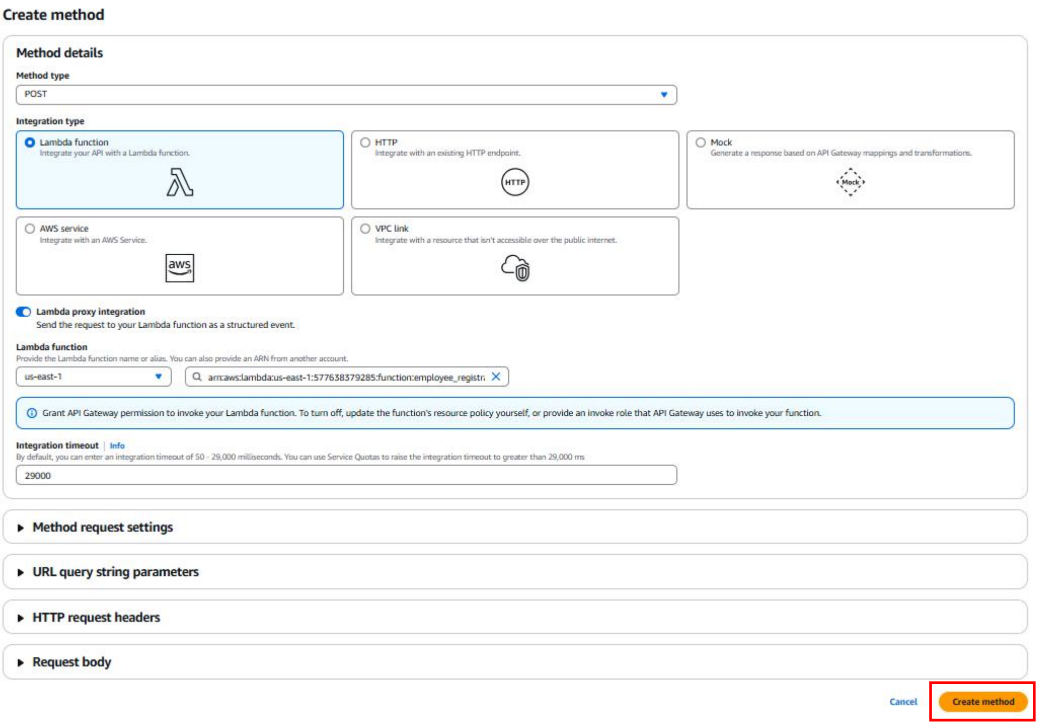
Resource details

○ Proxy resource Info
Proxy resources handle requests to all sub-resources. To create a proxy resource use a path parameter that ends with a plus sign, for example {proxy+}.

Resource path
/

Resource name
register

☑ CORS (Cross Origin Resource Sharing) Info
Create an OPTIONS method that allows all origins, all methods, and several common headers.



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 (on) — 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:employee_registr

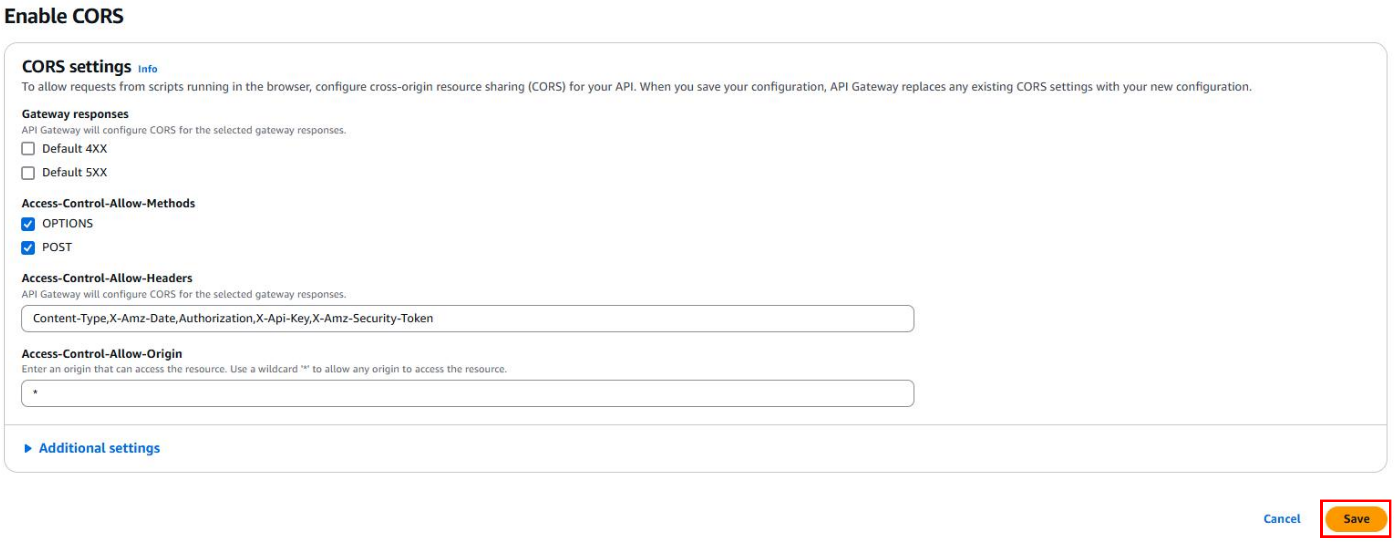
Grant API Gateway permission to invoke your Lambda function. To turn off, update the function's resource policy yourself, or provide an invoke role that API Gateway uses to invoke your function.

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

We create a POST method on 'register' and integrate it with our lambda function

Then, we enable CORS:



Enable CORS

CORS settings Info

To allow requests from scripts running in the browser, configure cross-origin resource sharing (CORS) for your API. When you save your configuration, API Gateway replaces any existing CORS settings with your new configuration.

Gateway responses — API Gateway will configure CORS for the selected gateway responses.

□ Default 4XX

□ Default 5XX

Access-Control-Allow-Methods

☑ OPTIONS

☑ POST

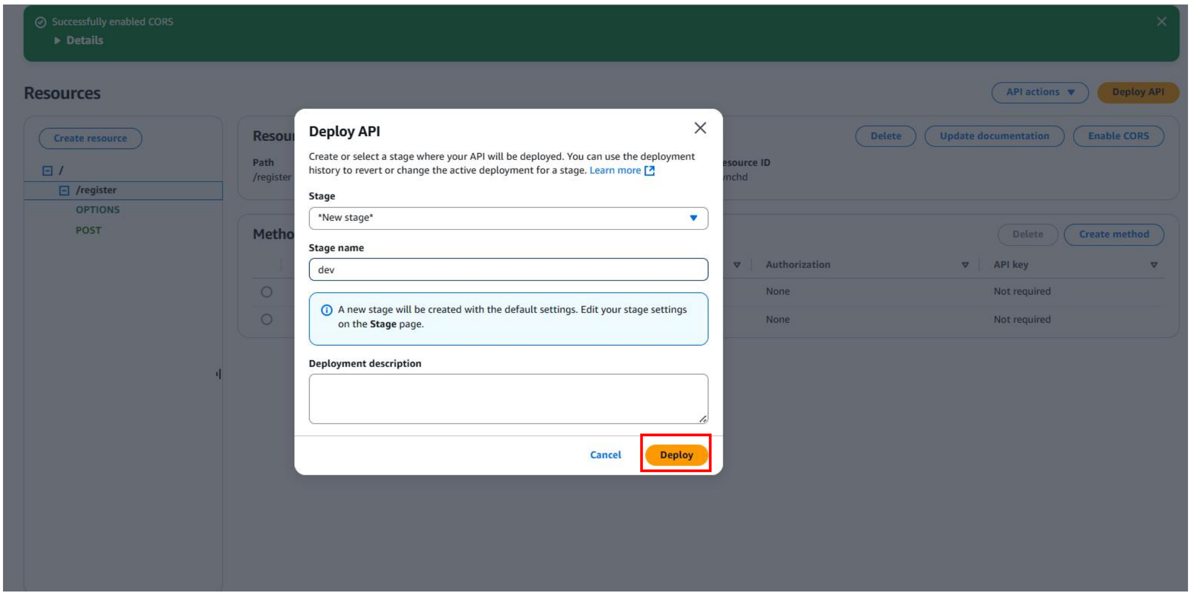
Access-Control-Allow-Headers — API Gateway will configure CORS for the selected gateway responses.

Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token

Access-Control-Allow-Origin — Enter an origin that can access the resource. Use a wildcard '*' to allow any origin to access the resource.

*

and we deploy the API:



Successfully enabled CORS

Deploy API

Create or select a stage where your API will be deployed. You can use the deployment history to revert or change the active deployment for a stage. Learn more

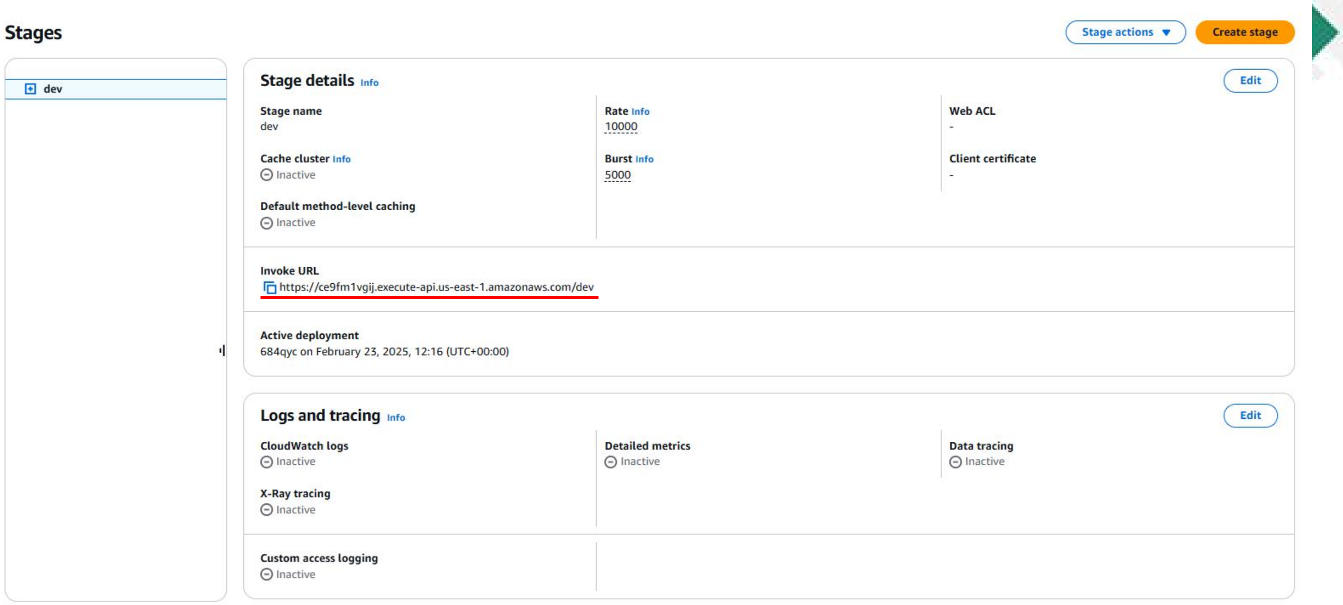
Stage: *New stage*

Stage name: dev

A new stage will be created with the default settings. Edit your stage settings on the **Stage** page.

Deployment description

After this we should get an invoke URL



Stages

Stage details Info

Stage name
dev

Rate Info
10000

Web ACL
-

Cache cluster Info
⊖ Inactive

Burst Info
5000

Client certificate
-

Default method-level caching
⊖ Inactive

Invoke URL
https://ce9fm1vgij.execute-api.us-east-1.amazonaws.com/dev

Active deployment
684qyc on February 23, 2025, 12:16 (UTC+00:00)

Logs and tracing Info

CloudWatch logs
⊖ Inactive

Detailed metrics
⊖ Inactive

Data tracing
⊖ Inactive

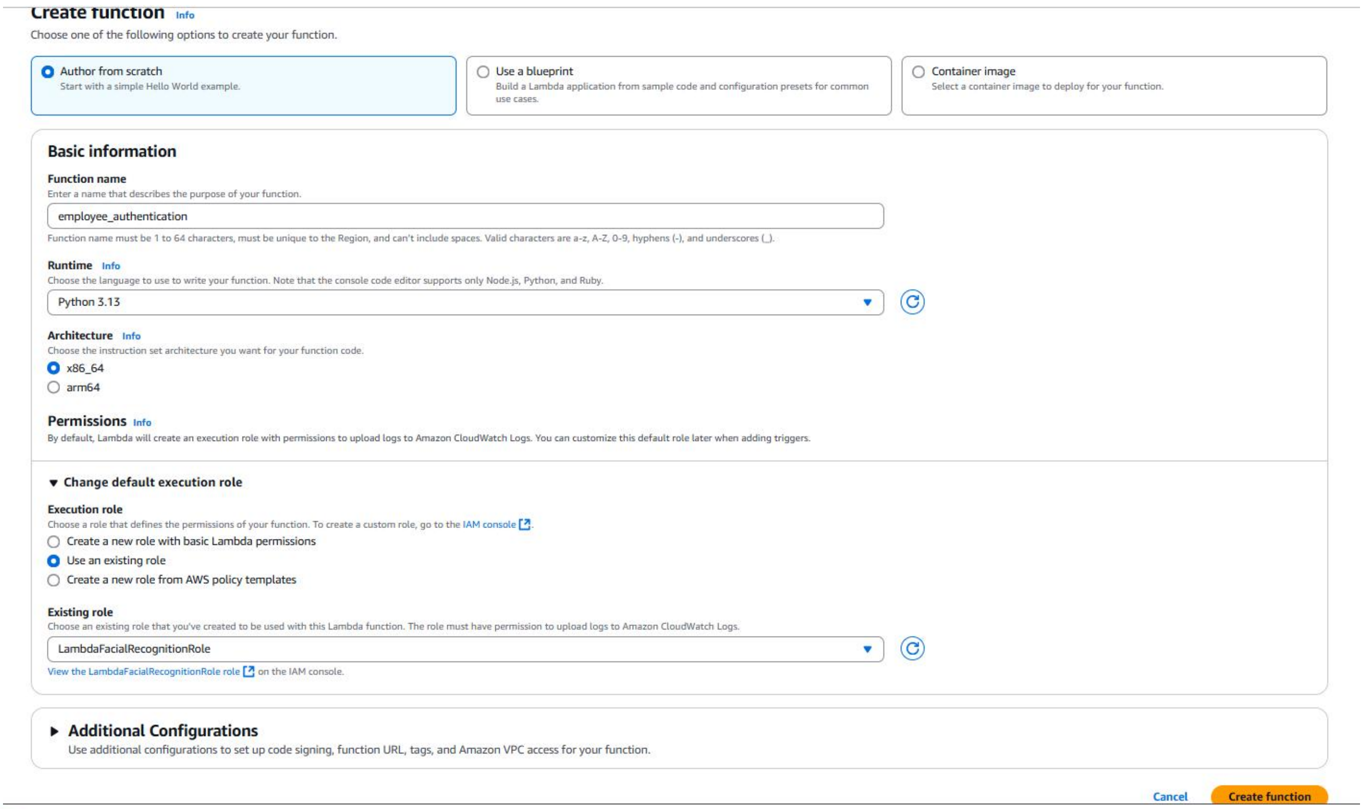
X-Ray tracing
⊖ Inactive

Custom access logging
⊖ Inactive

We'll paste in the variable 'apiEndpoint' in the **register.html** and **register.html** files before hosting them on S3.

Step3: create the employee_authentication lambda function and trigger it with an API gateway:

We go to lambda and we create a function named 'employee_authentication' without forgetting to add the **LambdaFacialRecognitionRole** created before



Create function Info

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.
employee_authentication
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.

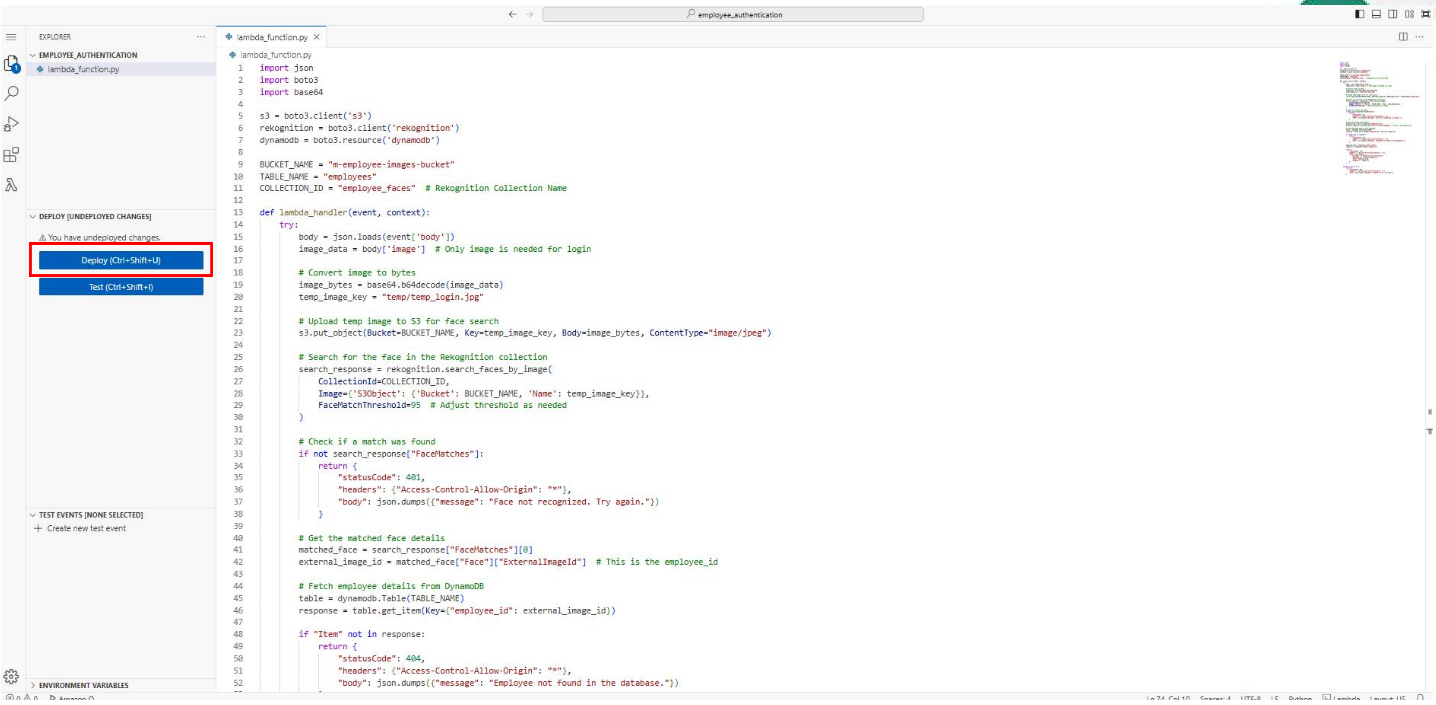
▼ Change default execution role

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.
LambdaFacialRecognitionRole
View the LambdaFacialRecognitionRole 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.

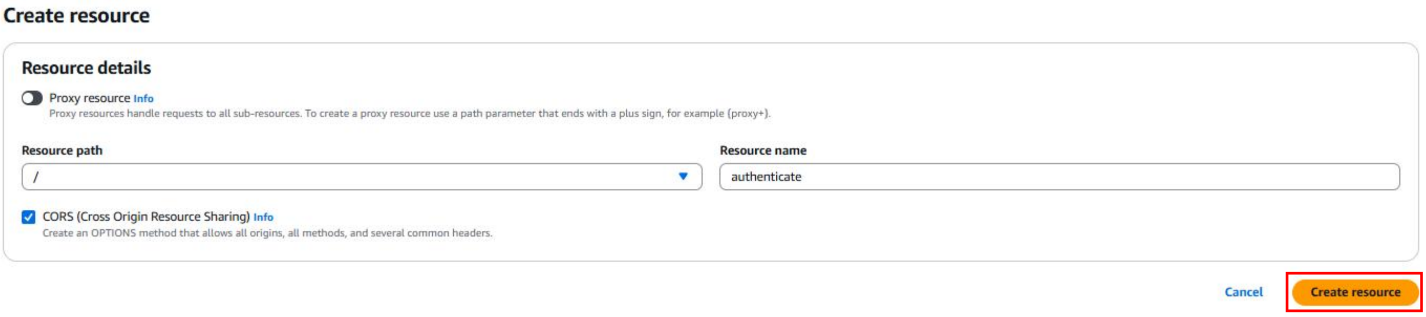
We click on ‘create function’, after that, we go to ‘code source’ and we type the following python code (the code is on the repository):



```python
import json
import boto3
import base64

s3 = boto3.client('s3')
rekognition = boto3.client('rekognition')
dynamodb = boto3.resource('dynamodb')

BUCKET_NAME = "m-employee-images-bucket"
TABLE_NAME = "employees"
COLLECTION_ID = "employee_faces"  # Rekognition Collection Name

def lambda_handler(event, context):
    try:
        body = json.loads(event['body'])
        image_data = body['image']  # Only image is needed for login

        # Convert image to bytes
        image_bytes = base64.b64decode(image_data)
        temp_image_key = "temp/temp_login.jpg"

        # Upload temp image to S3 for face search
        s3.put_object(Bucket=BUCKET_NAME, Key=temp_image_key, Body=image_bytes, ContentType="image/jpeg")

        # Search for the face in the Rekognition collection
        search_response = rekognition.search_faces_by_image(
            CollectionId=COLLECTION_ID,
            Image={'S3Object': {'Bucket': BUCKET_NAME, 'Name': temp_image_key}},
            FaceMatchThreshold=95  # Adjust threshold as needed
        )

        # Check if a match was found
        if not search_response["FaceMatches"]:
            return {
                "statusCode": 401,
                "headers": {"Access-Control-Allow-Origin": "*"},
                "body": json.dumps({"message": "Face not recognized. Try again."})
            }

        # Get the matched face details
        matched_face = search_response["FaceMatches"][0]
        external_image_id = matched_face["Face"]["ExternalImageId"]  # This is the employee_id

        # Fetch employee details from DynamoDB
        table = dynamodb.Table(TABLE_NAME)
        response = table.get_item(Key={"employee_id": external_image_id})

        if "Item" not in response:
            return {
                "statusCode": 404,
                "headers": {"Access-Control-Allow-Origin": "*"},
                "body": json.dumps({"message": "Employee not found in the database."})
```

We click on ‘Deploy’.

Then, we go to the API gateway and we create another resource named authenticate



Create resource

Resource details

○ Proxy resource Info

Proxy resources handle requests to all sub-resources. To create a proxy resource use a path parameter that ends with a plus sign, for example {proxy+}.

Resource path: /

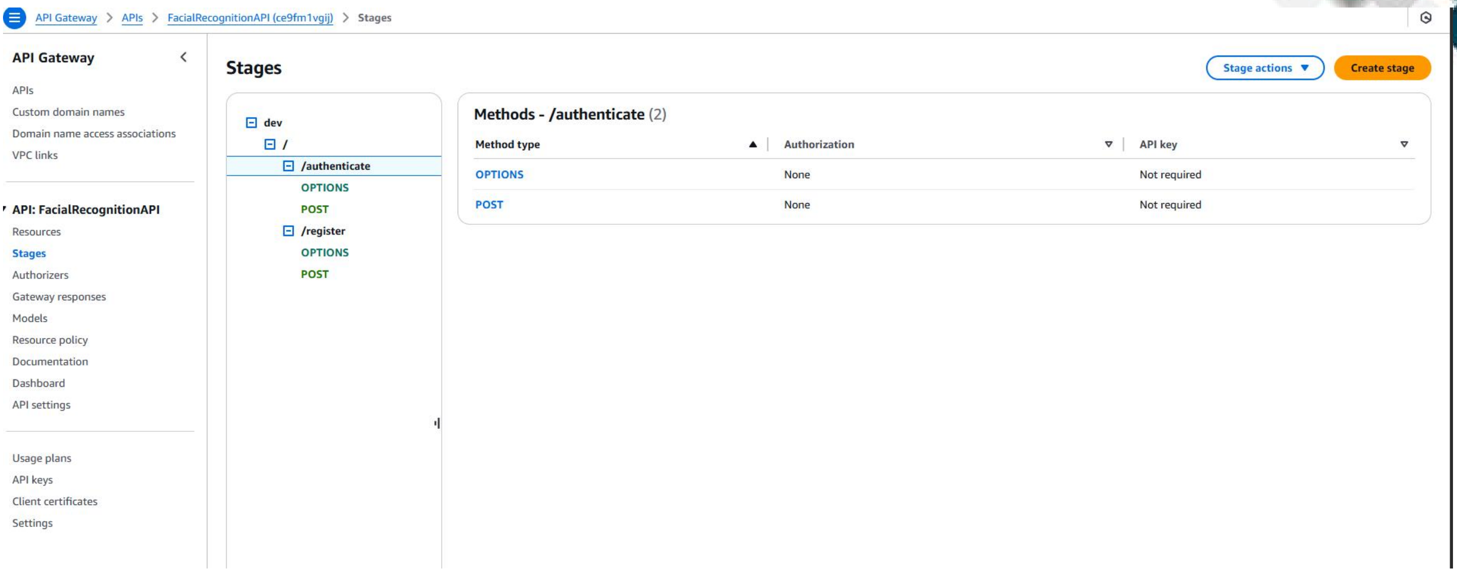
Resource name: authenticate

☑ CORS (Cross Origin Resource Sharing) Info

Create an OPTIONS method that allows all origins, all methods, and several common headers.

We do the same things we did on the register API except the POST method should be integrated with ‘employee_authentication’ lambda function.

We should get a result like this:



Stages

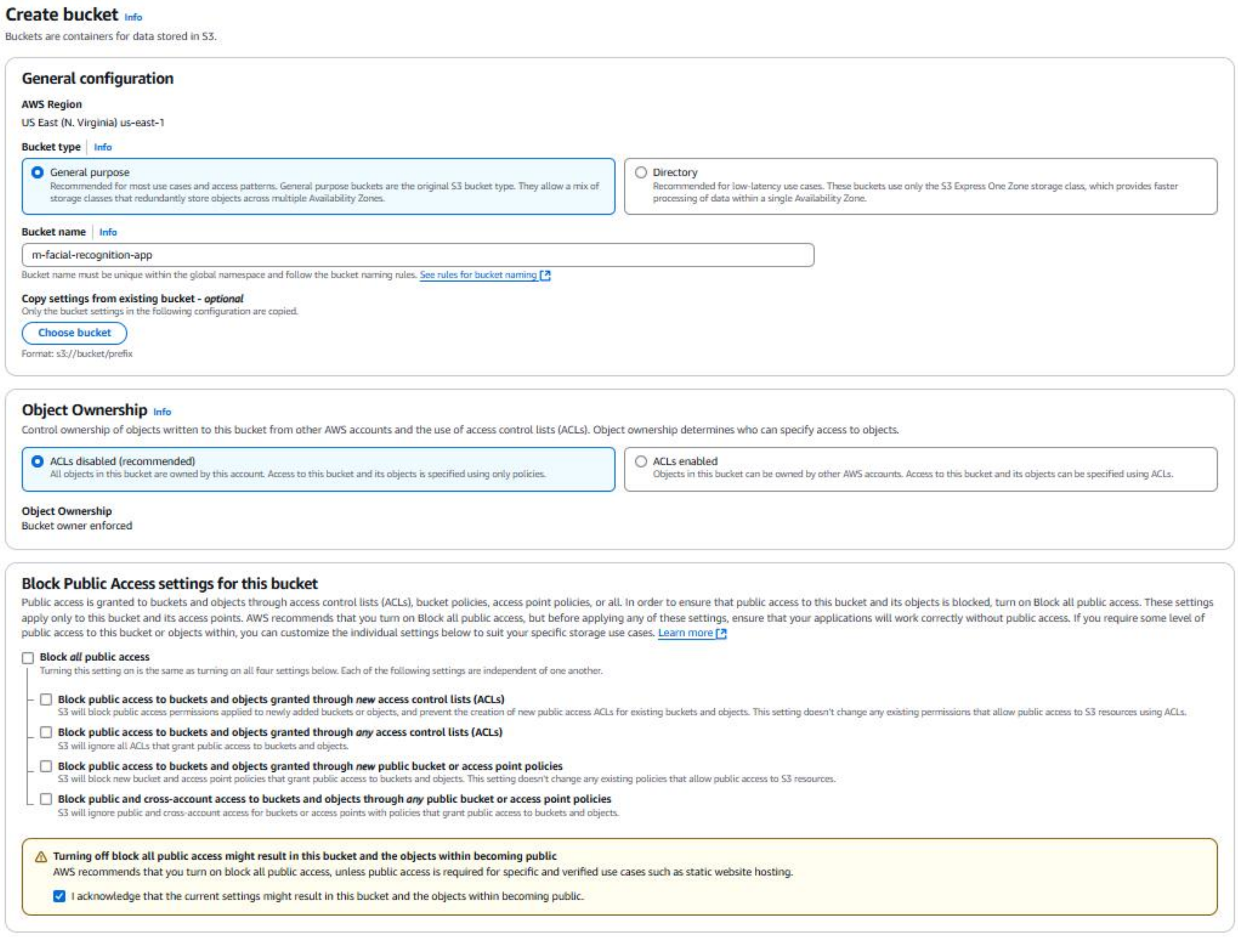
- dev
  - /
    - /authenticate
      - OPTIONS
      - POST
    - /register
      - OPTIONS
      - POST

Methods - /authenticate (2)

Method type	Authorization	API key
OPTIONS	None	Not required
POST	None	Not required

Step4: Hosting the 2 html files on S3:

After replacing the ‘apiEndpoint’ on the **register.html** file by **https://ce9fm1vgij.execute-api.us-east-1.amazonaws.com/dev/register** and the ‘apiEndpoint’ on **login.html** file by **https://ce9fm1vgij.execute-api.us-east-1.amazonaws.com/dev/authenticate** We host them on S3, we create a new bucket like it is shown below:



Create bucket Info

Buckets are containers for data stored in S3.

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
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

m-facial-recognition-app

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.

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

Block Public Access settings for this bucket

Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that public access to this bucket and its objects is blocked, turn on Block all public access. These settings apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying any of these settings, ensure that your applications will work correctly without public access. If you require some level of public access to this bucket or objects within, you can customize the individual settings below to suit your specific storage use cases. Learn more

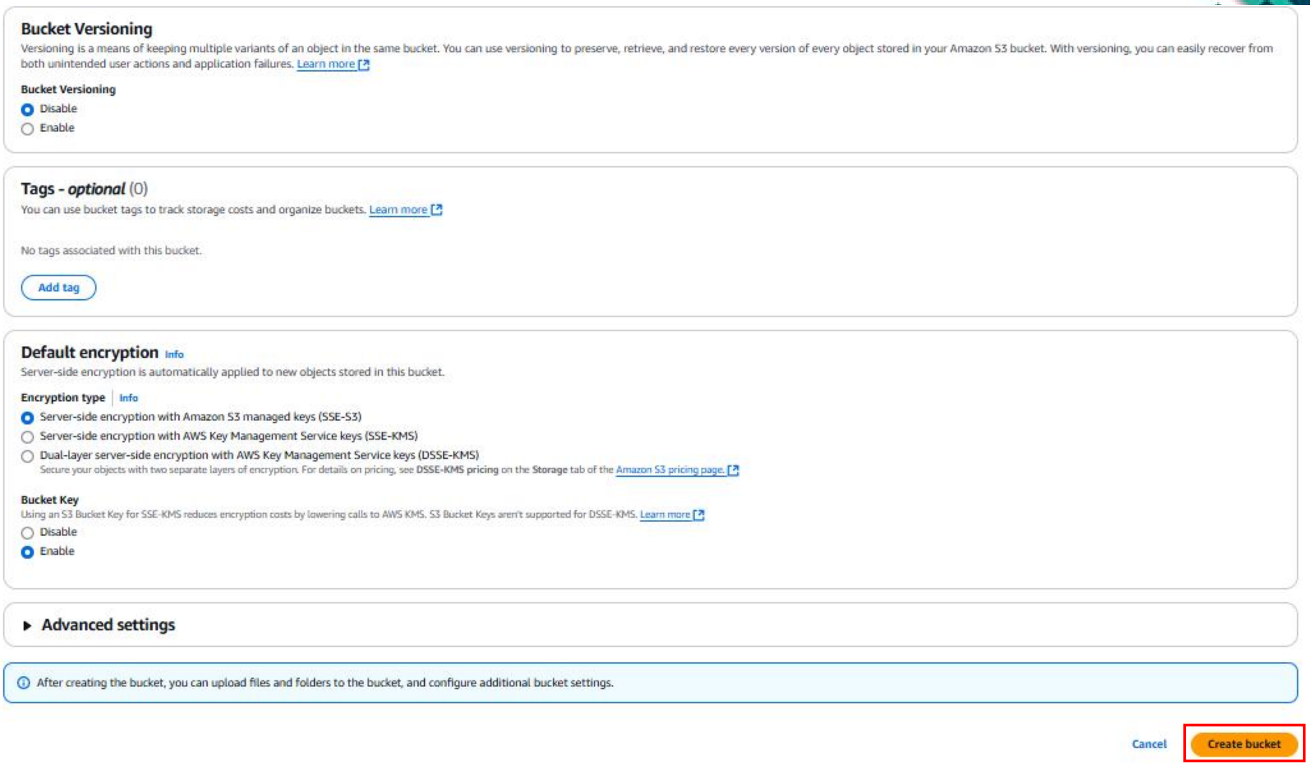
□ Block *all* public access
Turning this setting on is the same as turning on all four settings below. Each of the following settings are independent of one another.

- □ Block public access to buckets and objects granted through *new* access control lists (ACLs)
  S3 will block public access permissions applied to newly added buckets or objects, and prevent the creation of new public access ACLs for existing buckets and objects. This setting doesn't change any existing permissions that allow public access to S3 resources using ACLs.
- □ Block public access to buckets and objects granted through *any* access control lists (ACLs)
  S3 will ignore all ACLs that grant public access to buckets and objects.
- □ Block public access to buckets and objects granted through *new* public bucket or access point policies
  S3 will block new bucket and access point policies that grant public access to buckets and objects. This setting doesn't change any existing policies that allow public access to S3 resources.
- □ Block public and cross-account access to buckets and objects through *any* public bucket or access point policies
  S3 will ignore public and cross-account access for buckets or access points with policies that grant public access to buckets and objects.

⚠ Turning off block all public access might result in this bucket and the objects within becoming public

AWS recommends that you turn on block all public access, unless public access is required for specific and verified use cases such as static website hosting.

☑ I acknowledge that the current settings might result in this bucket and the objects within becoming public.



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.

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

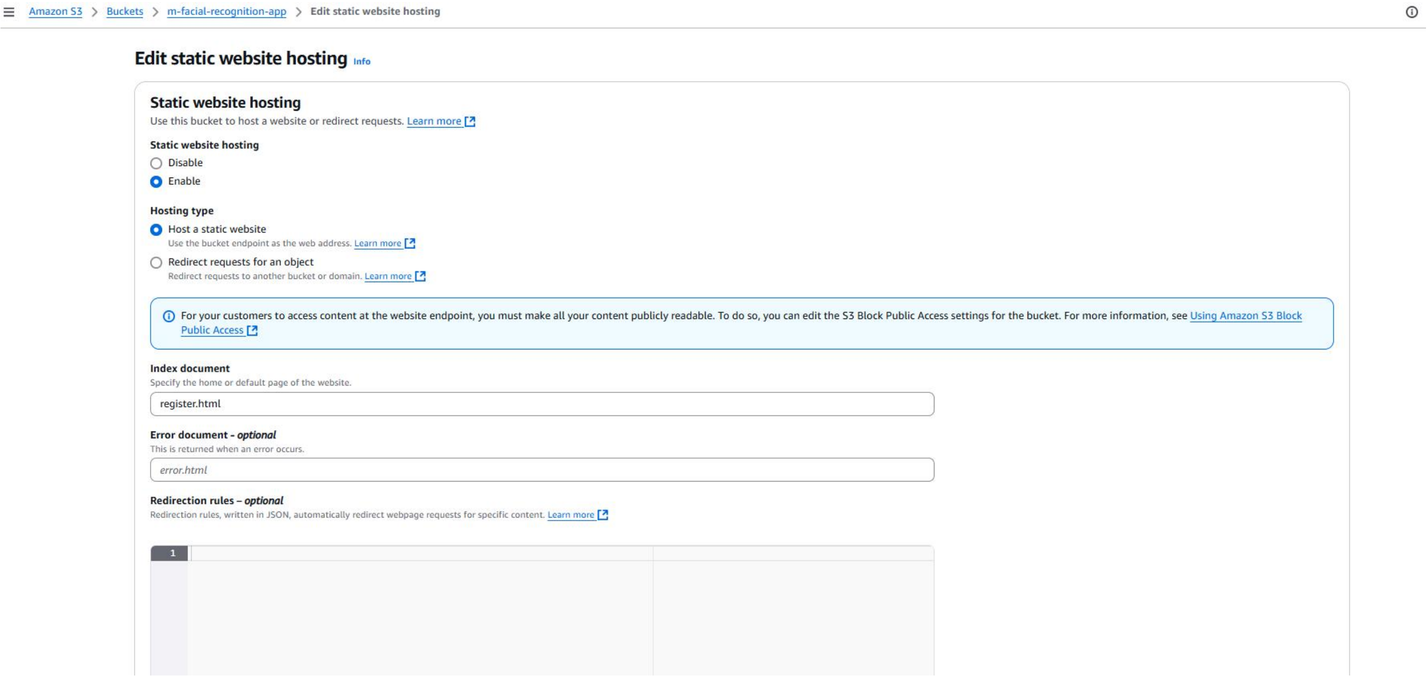
○ Disable

● Enable

▸ **Advanced settings**

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

We enable static website hosting, we go to properties section and all the way down to it and we click on 'edit', we fill it as it is shown below



Amazon S3 > Buckets > m-facial-recognition-app > Edit static website 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.

register.html

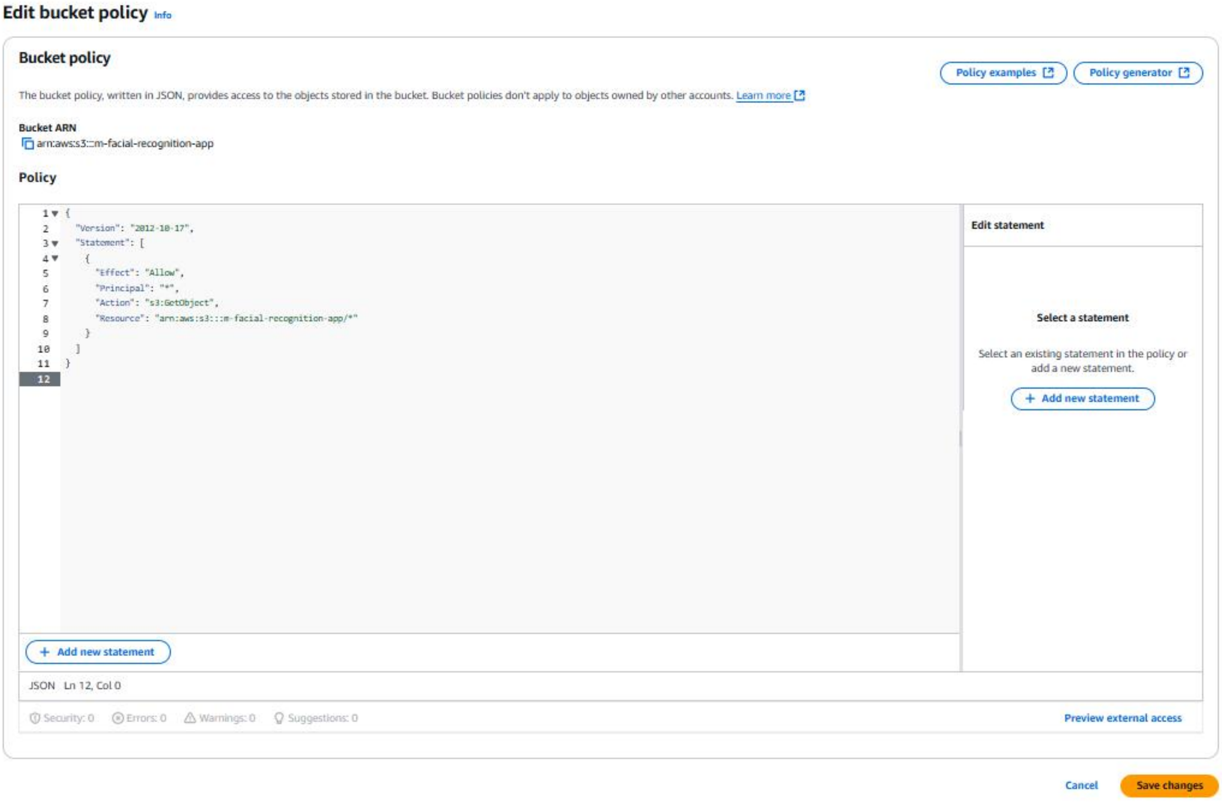
Error document - *optional*
This is returned when an error occurs.

error.html

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

After clicking on 'Save changes', we go to give it the following policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
   {
     "Effect": "Allow",
     "Principal": "*",
     "Action": "s3:GetObject",
     "Resource": "arn:aws:s3:::m-facial-recognition-app/*"
   }
  ]
}
```



Edit bucket policy Info

Bucket policy

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:::m-facial-recognition-app

Policy

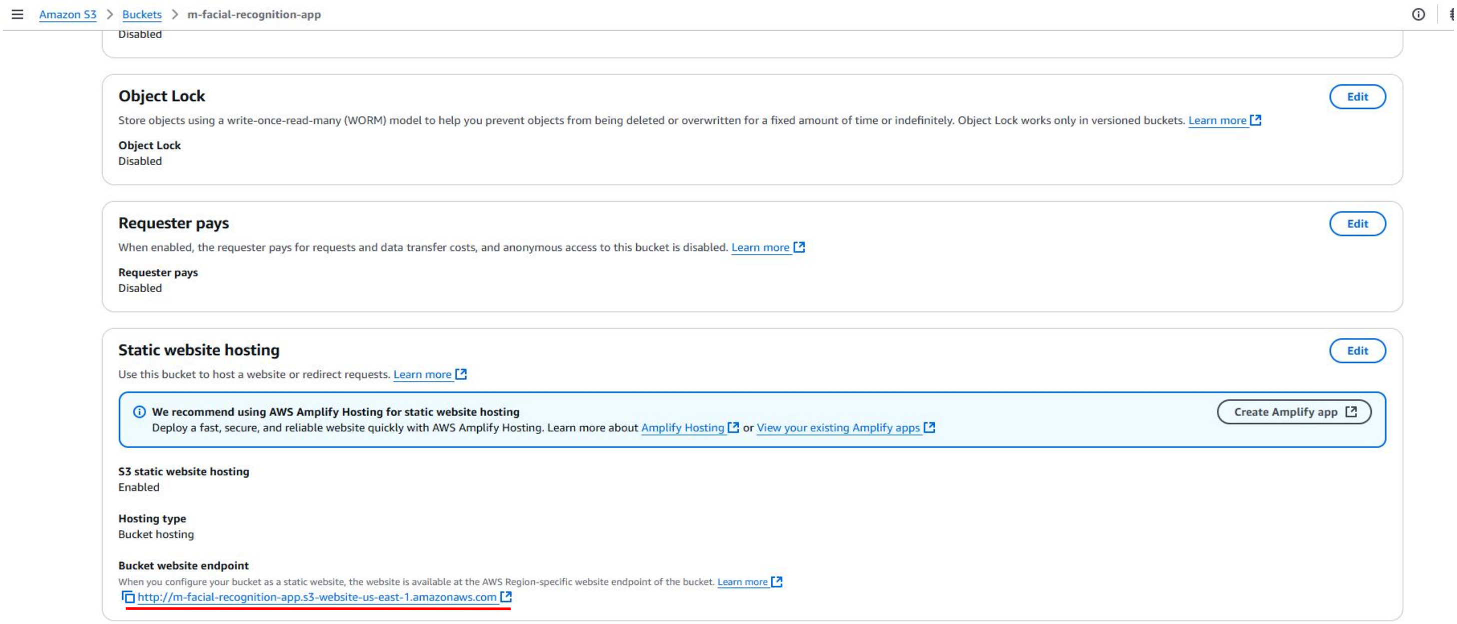
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::m-facial-recognition-app/*"
    }
  ]
}
```

JSON Ln 12, Col 0

Security: 0 Errors: 0 Warnings: 0 Suggestions: 0

We cick on ‘Save changes’

After uploading the **register.html** and the **login.html** files, we go now to get the link in the properties section:



Amazon S3 > Buckets > m-facial-recognition-app

Disabled

Object Lock

Store objects using a write-once-read-many (WORM) model to help you prevent objects from being deleted or overwritten for a fixed amount of time or indefinitely. Object Lock works only in versioned buckets. Learn more

Object Lock
Disabled

Requester pays

When enabled, the requester pays for requests and data transfer costs, and anonymous access to this bucket is disabled. Learn more

Requester pays
Disabled

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

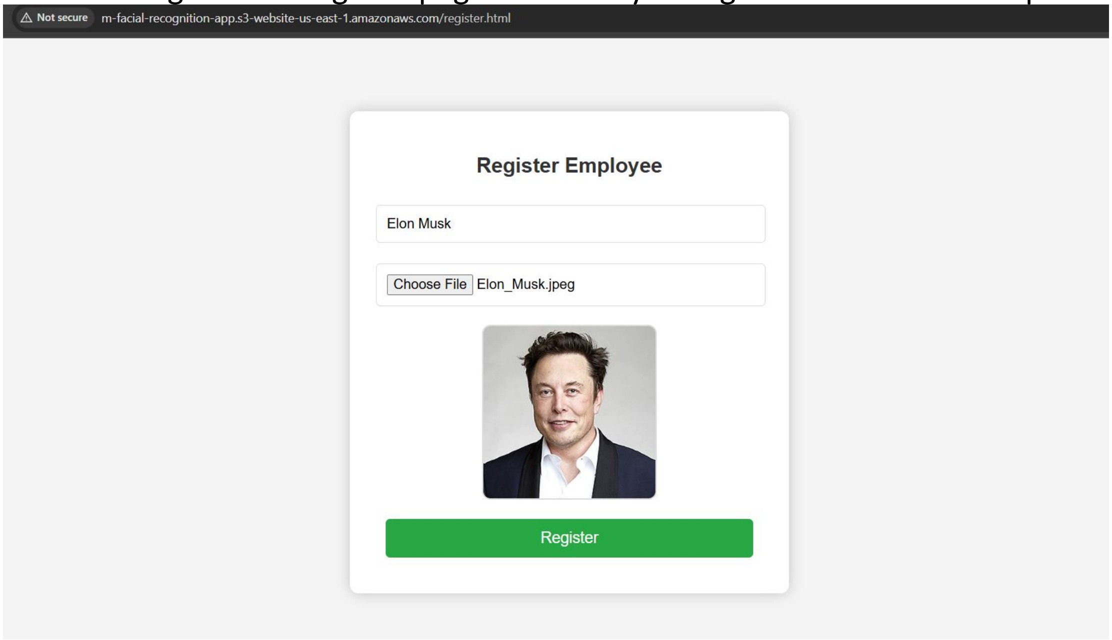
S3 static website hosting
Enabled

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://m-facial-recognition-app.s3-website-us-east-1.amazonaws.com

To go to login page we visit this link: http://m-facial-recognition-app.s3-website-us-east-1.amazonaws.com/register.html
To go to register page we visit this link: http://m-facial-recognition-app.s3-website-us-east-1.amazonaws.com/login.html

So now we go to the register page and we try to register a user for example:



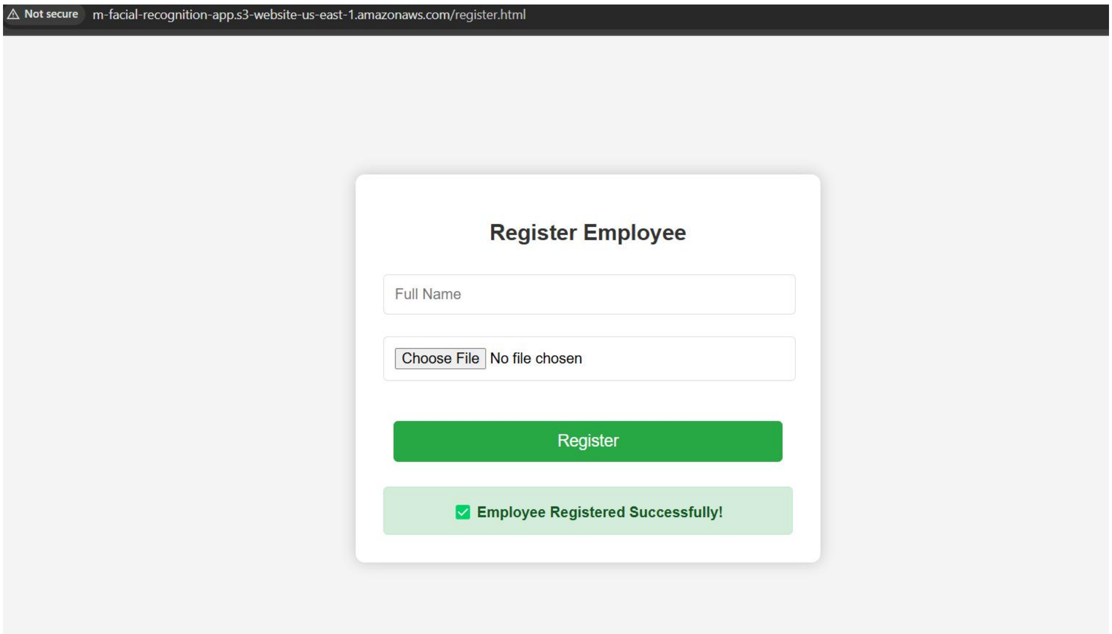
⚠ Not secure m-facial-recognition-app.s3-website-us-east-1.amazonaws.com/register.html

Register Employee

Elon Musk

Choose File Elon_Musk.jpeg

If we click on 'Register', we should get this



⚠ Not secure m-facial-recognition-app.s3-website-us-east-1.amazonaws.com/register.html

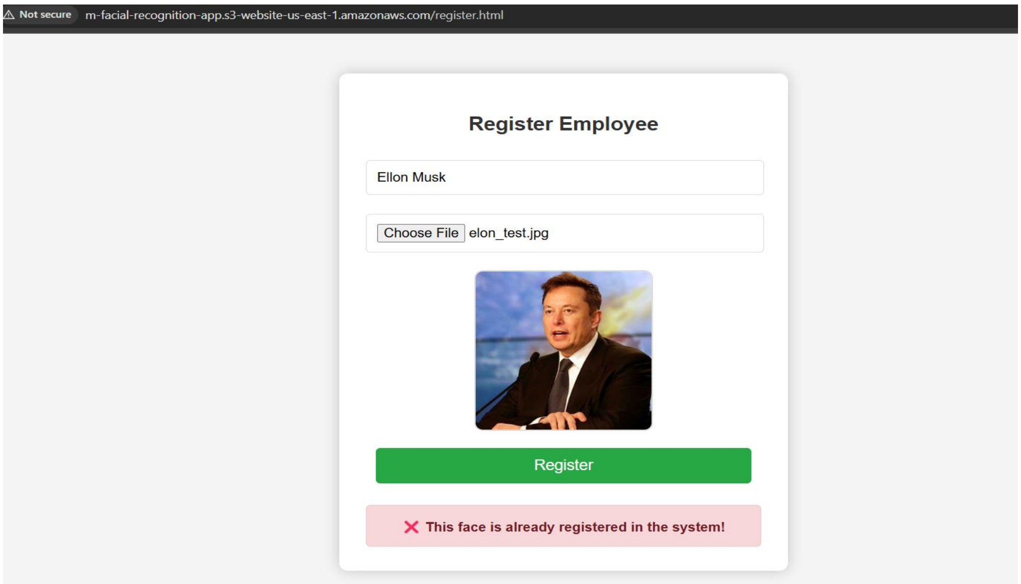
Register Employee

Full Name

Choose File No file chosen

✓ Employee Registered Successfully!

And if we try to register a different image of the same employee we get this



⚠ Not secure m-facial-recognition-app.s3-website-us-east-1.amazonaws.com/register.html

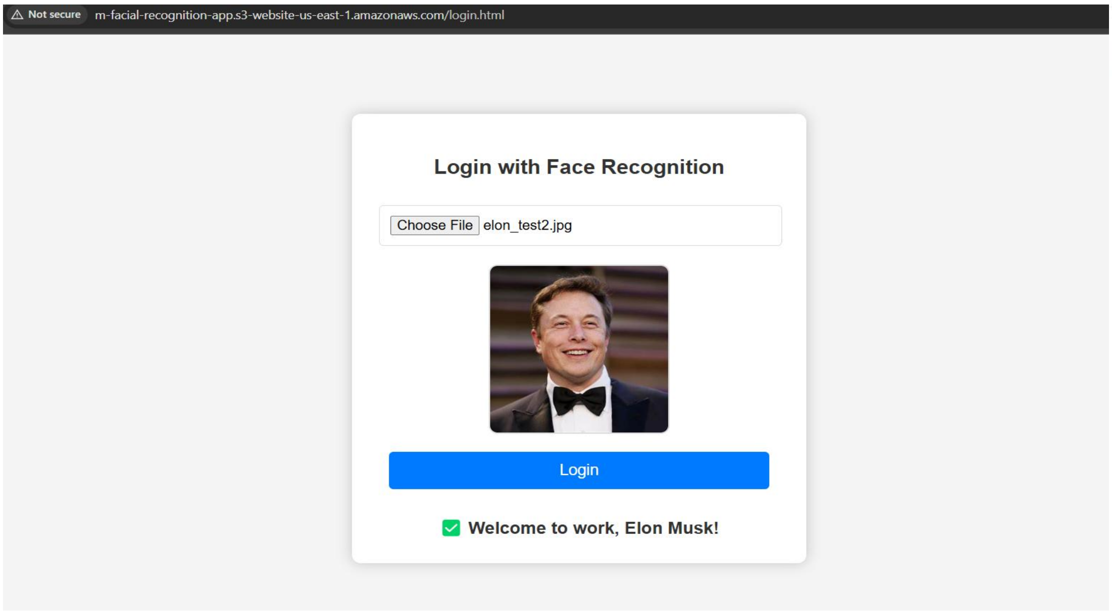
Register Employee

Ellon Musk

Choose File elon_test.jpg

✕ This face is already registered in the system!

If we try to login with a person already registered, we get this:



⚠ Not secure m-facial-recognition-app.s3-website-us-east-1.amazonaws.com/login.html

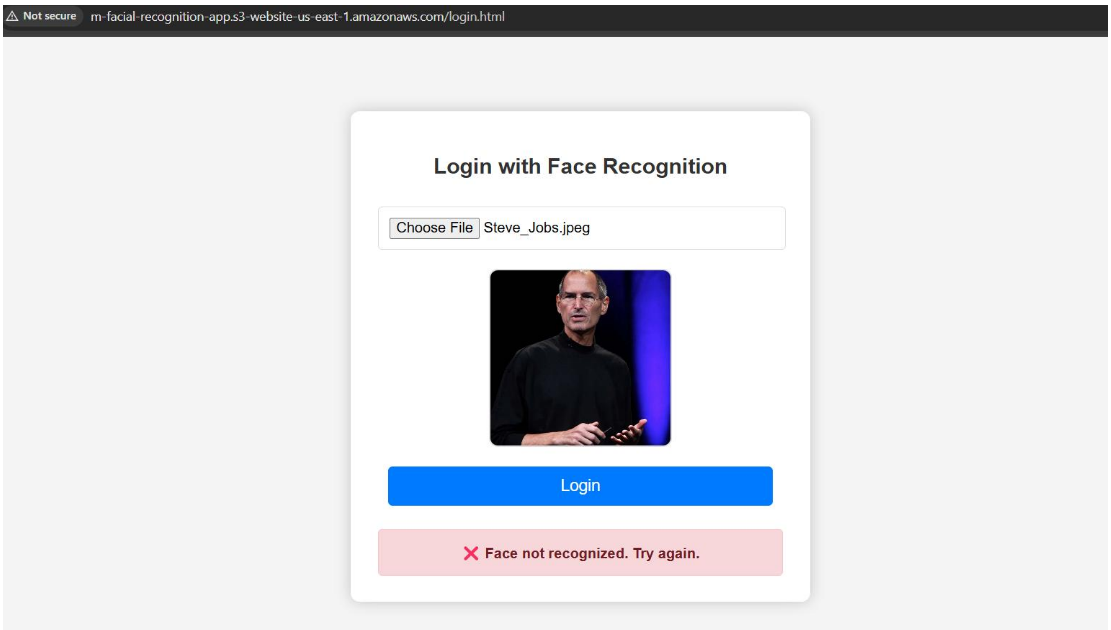
Login with Face Recognition

Choose File elon_test2.jpg

Login

✓ Welcome to work, Elon Musk!

If we try to login with a person not registered, we get this:



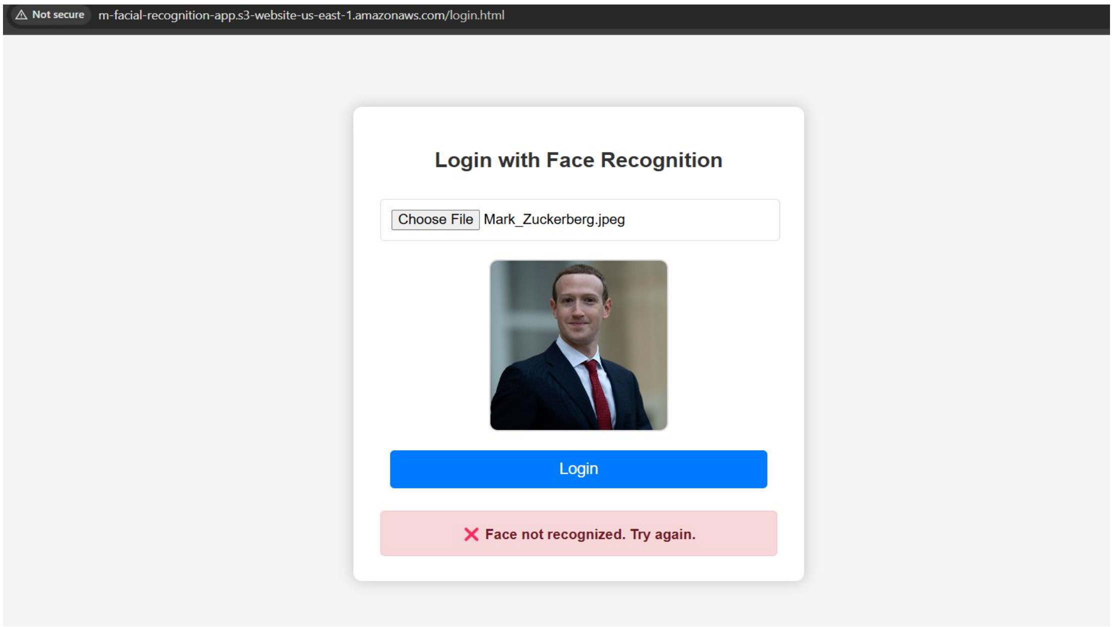
⚠ Not secure m-facial-recognition-app.s3-website-us-east-1.amazonaws.com/login.html

Login with Face Recognition

Choose File Steve_Jobs.jpeg

Login

✕ Face not recognized. Try again.



⚠ Not secure m-facial-recognition-app.s3-website-us-east-1.amazonaws.com/login.html

Login with Face Recognition

Choose File Mark_Zuckerberg.jpeg

Login

✕ Face not recognized. Try again.

Conclusion

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

Challenges Faced & Lessons Learned:

1) Preventing Duplicate Registrations with the Same Person’s Image

A major challenge was ensuring that two different users could not register using the same person’s image. This required implementing face comparison logic before storing new employees.
I developed a system that checks for existing faces in Rekognition Collections before registration, preventing unauthorized duplicate enrollments.

2) Managing S3 Bucket Permissions for Image Storage

Ensuring that images were stored securely while remaining accessible to AWS Rekognition and Lambda required fine-tuned S3 permissions. Without proper IAM policies, Rekognition and Lambda failed to access stored images.

3) DynamoDB Query Optimization for Fast Authentication

When an employee logs in, the system retrieves their data from DynamoDB based on the recognized face. Ensuring fast and efficient lookups was crucial, especially as the database grows.
I learned to structure DynamoDB tables effectively, use primary keys (employee_id) for quick lookups, and ensure that API responses remain fast and scalable.

Final Takeaways:

Through this project, I gained hands-on experience in serverless architecture and AWS facial recognition while building a real-world authentication system. Key skills I developed include:

✓ Implementing face recognition authentication using AWS Rekognition
✓ Configuring IAM roles and permissions for secure AWS service communication
✓ Handling S3 bucket policies for secure image storage and retrieval

This project has deepened my understanding of serverless cloud development, showcasing its advantages in terms of scalability, security, and cost-efficiency. Moving forward, I am excited to explore enhanced biometric security features, multi-factor authentication, and more advanced AI-driven recognition systems in cloud-based solutions.