Serverless Web Application Using AWS Lambda, API Gateway, S3, DynamoDB, and Cognito

Project Overview

The project is a **serverless web application** built entirely on AWS, leveraging managed services to reduce infrastructure overhead. The main features include:

1. **User Authentication**: Users can securely sign up and log in using **AWS Cognito**.
2. **File Upload**: Authenticated users can upload files directly to **S3**, with pre-signed URLs ensuring secure access.
3. **Metadata Management**: Each file's metadata (e.g., filename, user ID) is stored in **DynamoDB** for fast and scalable retrieval.
4. **Serverless API**: **API Gateway** routes requests to **Lambda functions**, which handle business logic like file metadata storage and pre-signed URL generation.
5. **Serverless Architecture**: The application is fully serverless, meaning it scales automatically without managing servers.

1. Introduction

This project demonstrates how to build a **serverless web application** using Amazon Web Services (AWS). The application allows:

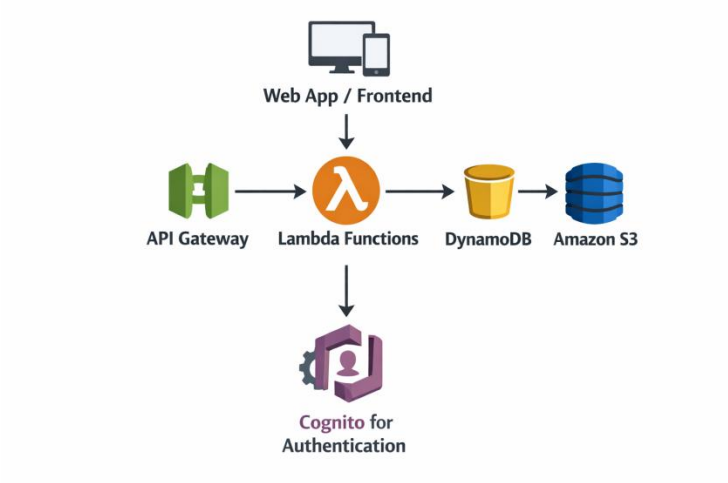
- User authentication using **Amazon Cognito**
- File upload using **Amazon S3**
- File metadata storage using **DynamoDB**
- REST APIs using **API Gateway**
- Business logic using **Lambda functions**

This architecture is fully serverless, scalable, and cost-efficient.

2. Architecture Overview

Flow Diagram (Conceptual)

1. User signs in via **Cognito User Pool**
2. User receives **JWT Token**
3. Frontend calls an **API Gateway Endpoint** with JWT
4. **Lambda** generates a presigned URL for file upload
5. User uploads file directly to **S3**
6. Another API stores metadata in **DynamoDB**



3. AWS Services Used

Service	Purpose
Amazon Cognito	User sign-up/login, authentication
Amazon API Gateway	Exposes REST API endpoints
AWS Lambda	Backend logic (presigned URL + metadata storage)
Amazon S3	Stores uploaded files
Amazon DynamoDB	Stores file metadata
IAM	Access control roles and policies

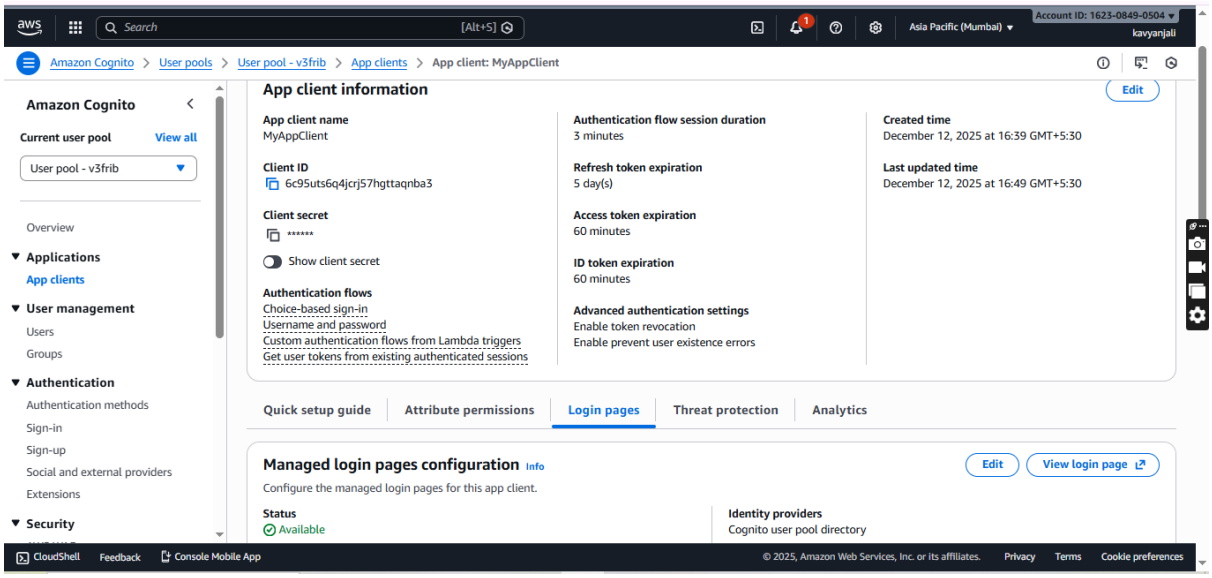
4. Step-by-Step Setup (AWS Console)

1) Create Cognito User Pool (Authentication)

1. Console → Services → **Cognito** → **Manage User Pools** → **Create a user pool**.
2. Choose **"Start from scratch"**.
3. Pool name: MyAppUserPool → **Next**.
4. **Users and sign-in**: Choose username or email as the sign-in method (most apps use **Email**). Select "Email" if you want email verification. → **Next**.
5. **Security**: Choose password policies as desired (default is okay). Enable email verification if required.
6. **App clients**: Under **App clients** → **Add an app client**:
    - App client name: MyAppClient
    - Leave "Generate client secret" unchecked for browser apps (if you plan a pure SPA).
    - Create client.
7. **App client settings (Hosted UI / OAuth)**:
    - After pool created → App integration → **App client settings**.
    - Select the app client MyAppClient.
    - Callback URL(s): add your frontend URL (e.g., http://localhost:8000 for dev).
    - Sign out URL(s): same or your sign-out page.
    - OAuth 2.0 flows: choose **Authorization code grant** (or **Implicit** for SPAs though it's less recommended).

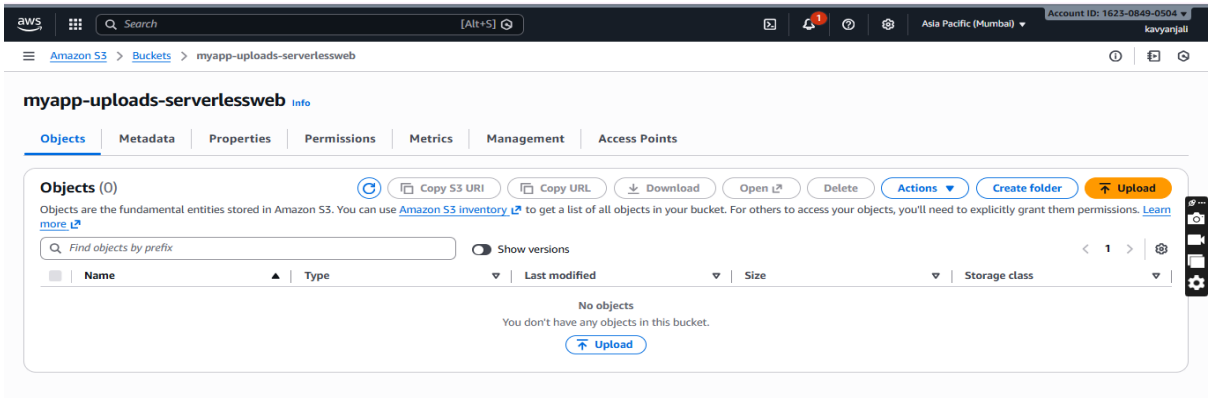
- OAuth scopes: email, openid, profile.
- Save changes.

8. **Domain**: App integration → Domain name → pick a domain prefix (or use your custom domain).
9. Note the **User Pool Id**, **App client id**, and **Cognito domain** — you'll need these in the frontend and for API Gateway authorizer.



2) Create S3 bucket (File storage)

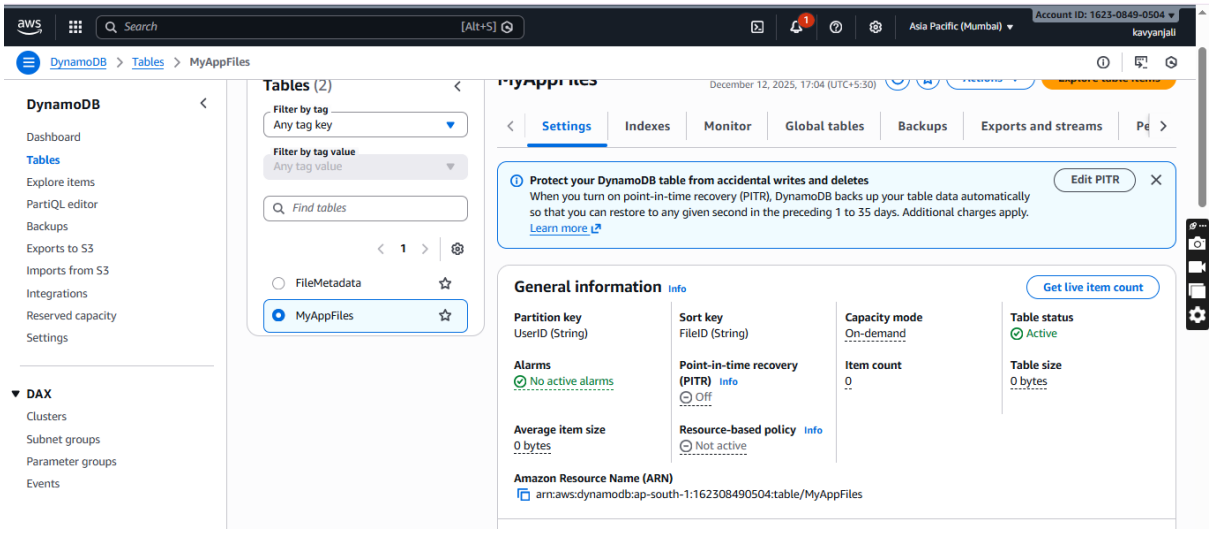
1. Console → Services → **S3** → **Create bucket**.
2. Bucket name: myapp-uploads-serverlessweb Region: pick your preferred region.
3. Uncheck "Block all public access"? NO — keep **public access blocked**. Presigned URLs allow upload without opening bucket publicly.
4. Versioning: optional (helpful for recovery).
5. Server-side encryption: enable (SSE-S3 or SSE-KMS) — recommended.
6. Create the bucket.
7. (Optional) Add lifecycle rules to expire or transition objects (good for cost control).



3) Create DynamoDB table (Metadata)

1. Console → **DynamoDB** → **Create table**.
2. Table name: MyAppFiles
3. Primary key:
   - Partition key: UserId (String)
   - Sort key: FileId (String) — you can use a UUID
4. Capacity mode: **On-demand** is easiest (no capacity planning).

5. Create table.
6. (Optional) Add a Global Secondary Index (e.g., FileTypeIndex) if you need search patterns.



4) Create IAM Roles & Policies for Lambda

You'll need a role that allows Lambda to call S3 and DynamoDB.

A. Create policy for presigned URL Lambda

1. Console → **IAM** → **Policies** → **Create policy** → JSON tab.
2. Paste (adjust bucket and region names):

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Sid": "S3PresignAccess",
   "Effect": "Allow",
   "Action": [
    "s3:PutObject",
    "s3:PutObjectAcl",
    "s3:GetObject"
   ],
   "Resource": " arn:aws:s3:::myapp-uploads-serverlessweb"
  }
 ]
}
```

3. Name: MyAppPresignPolicy → Create policy.

B. Create policy for metadata Lambda

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Sid": "DDBWriteAccess",
   "Effect": "Allow",
   "Action": [
    "dynamodb:PutItem",
```

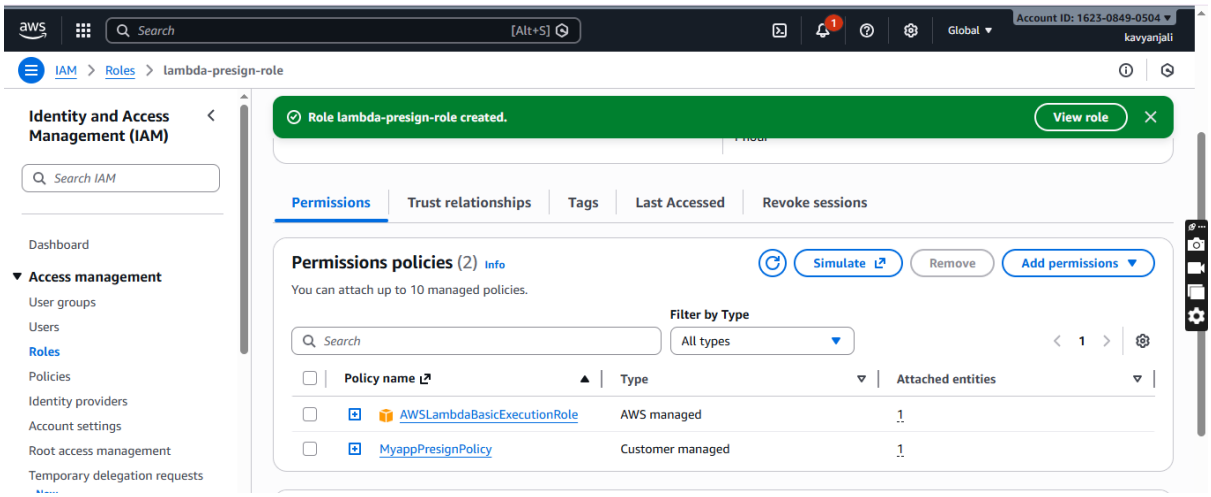
```
      "dynamodb:UpdateItem",
      "dynamodb:GetItem",
      "dynamodb:Query"
    ],
    "Resource": " arn:aws:dynamodb:ap-south-1:162308490504:table/MyAppFiles"
  }
 ]
}
```

Name: MyAppDDBPolicy.

C. Create Lambda execution roles

1. Console → **IAM** → **Roles** → **Create role**.
2. Choose **Lambda** as trusted entity.
3. Attach AWSLambdaBasicExecutionRole + MyAppPresignPolicy (for presign Lambda).
4. Name: lambda-presign-role.
5. Repeat creating lambda-metadata-role attaching AWSLambdaBasicExecutionRole + MyAppDDBPolicy.

Note: you can also use a single role for both Lambdas if preferred (attach both policies).



5) Create Lambda: Generate Presigned URL

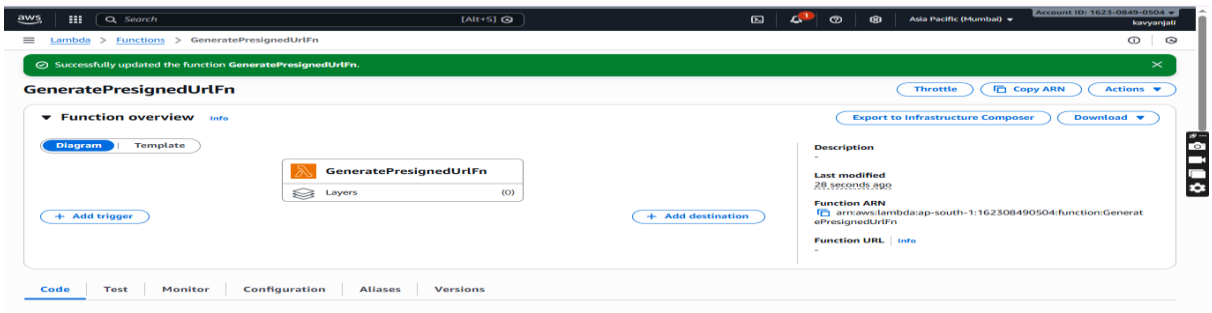
1. Console → **Lambda** → **Create function** → **Author from scratch**.
   - Function name: GeneratePresignedUrlFn
   - Runtime: **Python 3.11**
   - Execution role: **Use an existing role** → select lambda-presign-role.
2. Create function.
3. Replace the default handler code with the sample below (Python):

Python Lambda (presign generator) — lambda_function.py

```
import os
import json
import boto3
import uuid
import time

s3 = boto3.client('s3')
BUCKET = os.environ.get('BUCKET_NAME')  # set in Lambda env vars
```

```
def lambda_handler(event, context):
    # Expect JSON body with 'fileName' and 'contentType' and 'userId'
    body = event.get('body')
    if isinstance(body, str):
        body = json.loads(body)
    file_name = body.get('fileName')
    content_type = body.get('contentType', 'application/octet-stream')
    user_id = body.get('userId', 'anonymous')

    # Create a unique key (prefix by user for organization)
    file_id = str(uuid.uuid4())
    key = f"{user_id}/{file_id}_{file_name}"

    # Generate presigned URL
    presigned = s3.generate_presigned_url(
        'put_object',
        Params={
            'Bucket': BUCKET,
            'Key': key,
            'ContentType': content_type
        },
        ExpiresIn=900  # 15 minutes
    )

    response = {
        "uploadUrl": presigned,
        "key": key,
        "fileId": file_id
    }

    return {
        "statusCode": 200,
        "headers": {"Content-Type": "application/json", "Access-Control-Allow-Origin": "*"},
        "body": json.dumps(response)
    }
```

4. In **Configuration → Environment variables** add BUCKET_NAME = myapp-uploads-serverlessweb
5. Increase timeout to e.g., 10 seconds (not necessary but harmless).
6. Save.



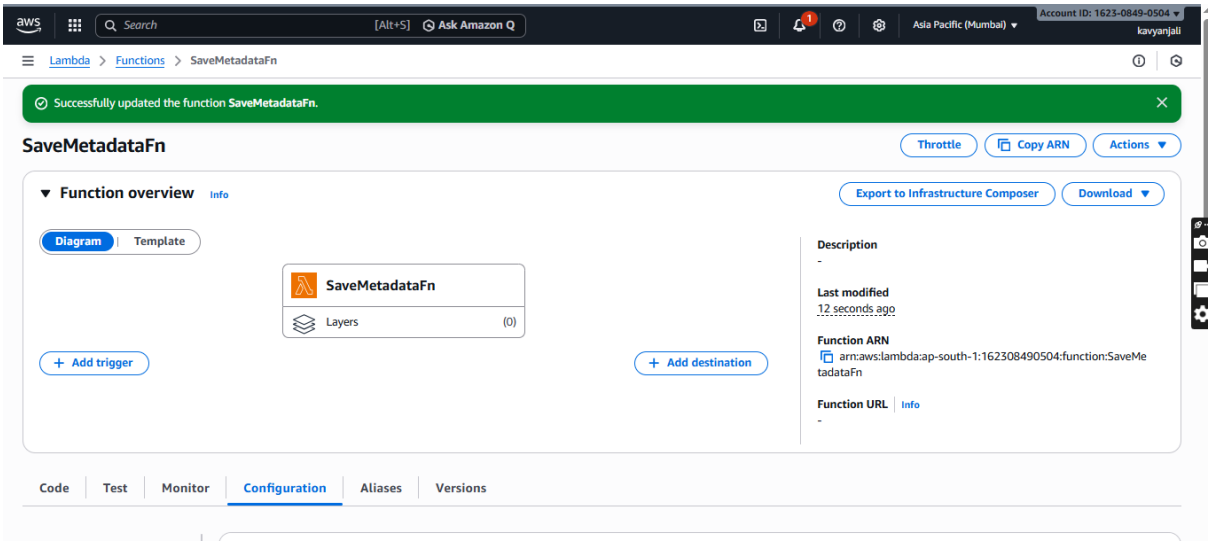
6) Create Lambda: Save metadata to DynamoDB

1. Console → **Lambda** → **Create function**.
    - Function name: SaveMetadataFn
    - Runtime: Python 3.11
    - Role: lambda-metadata-role
2. Add code:

Python Lambda (save metadata) — lambda_function.py

```
import os
import json
import boto3
from datetime import datetime

dynamodb = boto3.resource('dynamodb')
TABLE_NAME = os.environ.get('TABLE_NAME')
table = dynamodb.Table(TABLE_NAME)

def lambda_handler(event, context):
    body = event.get('body')
    if isinstance(body, str):
        body = json.loads(body)

    user_id = body.get('userId')
    file_id = body.get('fileId')
    key = body.get('key')
    size = body.get('size')
    content_type = body.get('contentType')
    extra = body.get('extra', {})

    if not (user_id and file_id and key):
        return {"statusCode":400, "body": json.dumps({"error":"missing fields"})}

    item = {
        "UserId": user_id,
        "FileId": file_id,
        "S3Key": key,
        "Size": size or 0,
        "ContentType": content_type or "application/octet-stream",
        "UploadedAt": datetime.utcnow().isoformat(),
        "Extra": extra
    }

    table.put_item(Item=item)

    return {
        "statusCode":200,
        "headers":{"Content-Type":"application/json","Access-Control-Allow-Origin":"*"},
        "body": json.dumps({"status":"ok", "item": item})
    }
```

3. Set environment variable TABLE_NAME = MyAppFiles.
4. Save.

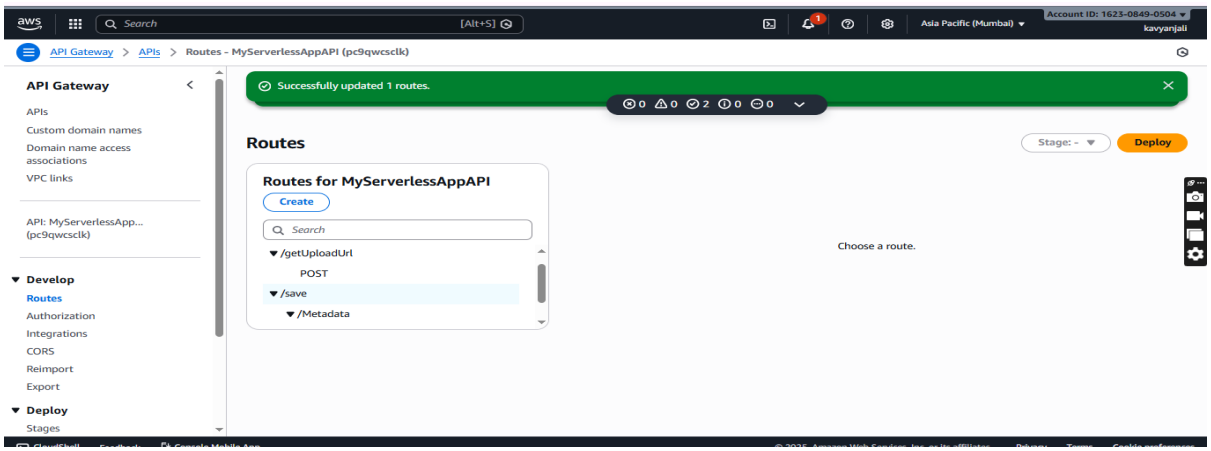


7) Create API Gateway (HTTP API) & secure with Cognito authorizer

We'll create an HTTP API (cheaper and simpler than REST API) and attach Cognito Authorizer.

1. Console → **API Gateway** → **Create API** → **HTTP API** → Build.
2. **Configure routes**
   - Add route POST /getUploadUrl
   - Add route POST /saveMetadata
3. **Integrations**
   - For each route, create integration → choose **Lambda function** → pick the corresponding Lambda (GeneratePresignedUrlFn for /getUploadUrl, SaveMetadataFn for /saveMetadata).
4. **Authorization (Cognito)**
   - On left → **Authorizers** → **Create and configure an authorizer**.
   - Type: **Cognito**.
   - Name: CognitoAuth.
   - Select the User Pool MyAppUserPool and App client MyAppClient.
   - Save.
5. Attach the authorizer to both routes:
   - Routes → select /getUploadUrl → Edit → Authorization → choose CognitoAuth (require authorization).
   - Do same for /saveMetadata.
6. **CORS**
   - Set CORS configuration for the API (enable origin http://localhost:8000 and allowed methods POST, OPTIONS).
7. **Deploy**
   - Review and **Deploy**. Note the **Invoke URL**.

Note: API Gateway HTTP API supports automatic JWT authorizer for Cognito. When the client authenticates with Cognito, it obtains an id_token or access_token. Send the token in Authorization: Bearer <token> header.

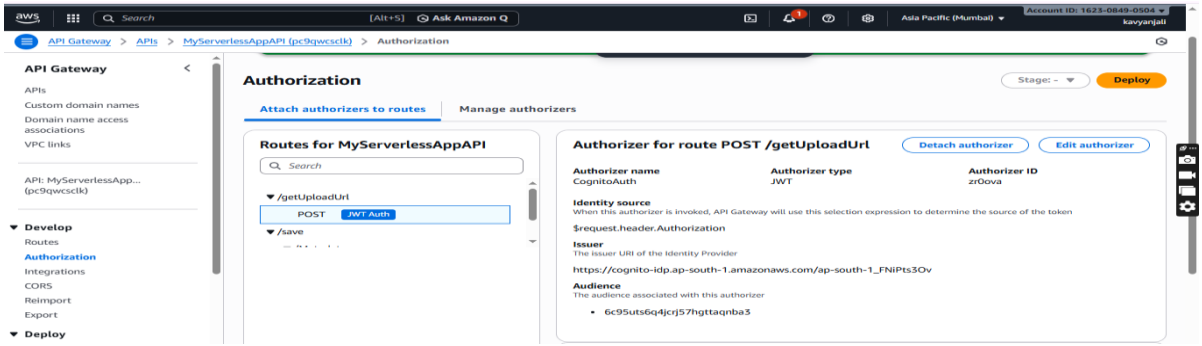


Steps (HTTP API)

1. AWS Console → **API Gateway** → **APIs** → click your HTTP API (e.g., MyServerlessAppAPI).
2. In the left menu look for **Authorization** or **Authorizers**.
    - If you see **Authorization** in the left nav, click it.
    - If you only see high-level menu items (Integrations, CORS, Routes), click **Routes** first, then click the name of the API (top-left) — the left-hand nav should change and show **Authorization**.
3. Click **Create** (or **Create authorizer**).
4. For **Authorizer type** choose **Cognito** if available — otherwise choose **JWT** (both work; JWT is the generic one).
    - **If you choose Cognito**: select the **User pool** dropdown MyAppUserPool and the **App client** MyAppClient. Save.
    - **If you choose JWT**: fill these fields:
        - **Name**: CognitoAuth
        - **Issuer**: https://cognito-idp.ap-south-1.amazonaws.com/ap-south-1_FNiPts3Ov **Audience** (one or more): the App client id (example: 3k1a2b3c4d5e6f7g8h9i0j) — this is the client id (not secret).
        - Save the authorizer.
5. After saving, go to **Routes**.
6. Select a route (e.g., POST /getUploadUrl). Click **Attach authorizer** (or Edit → Authorization).
7. Choose CognitoAuth (or the JWT authorizer you created) and set **Authorization required** (true).
8. Save changes.

Notes

- For HTTP APIs the authorizer validates JWTs (id_token or access_token). Use Authorization: Bearer <id_token> header from Cognito hosted UI.
- HTTP APIs do **not** require a manual “Deploy” step like REST APIs; changes are live once saved.

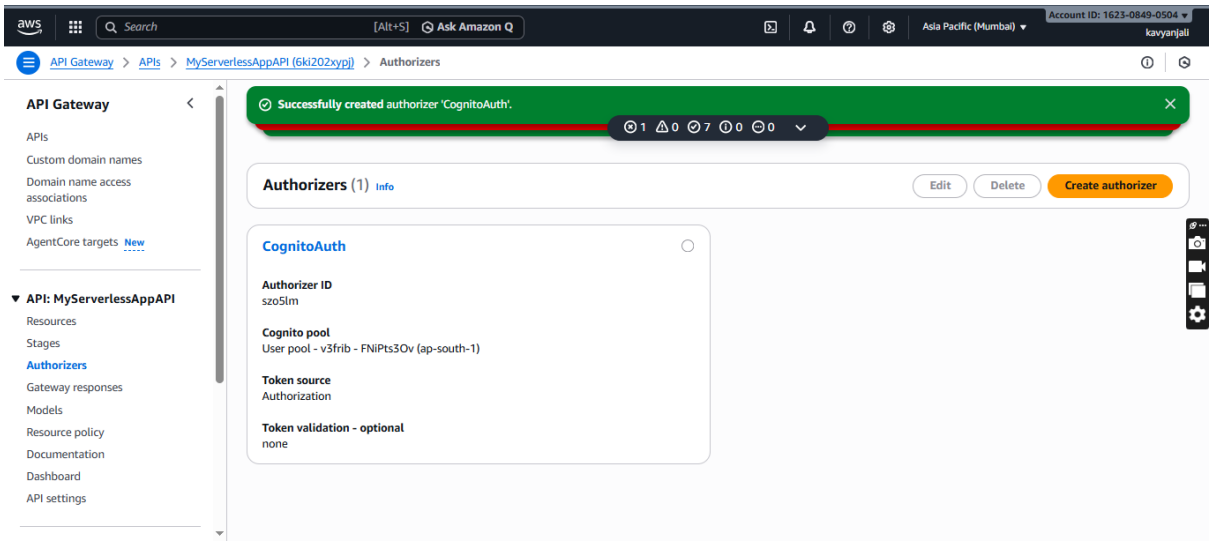


B — If your API is a REST API (older)

The REST API workflow shows **Authorizers** under the API configuration page. You create a Cognito authorizer and attach it to methods.

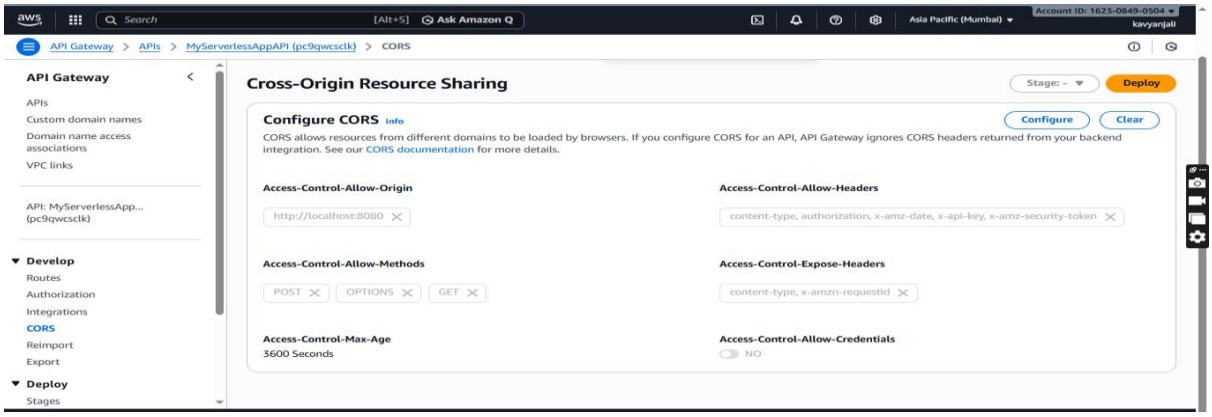
Steps (REST API)

1. Console → **API Gateway** → **APIs** → click your **REST API**.
2. In left nav click **Authorizers**.
3. Click **Create New Authorizer** → choose **Cognito**.
4. Name it CognitoAuth, select your **User Pool** MyAppUserPool and **App client** MyAppClient.
5. Save.
6. In left nav click **Resources**, select the resource/method (e.g., POST /getUploadUrl → POST).
7. Click **Method Request** → **Authorization** → choose CognitoAuth.
8. **Deploy** the API (Actions → Deploy API → select Stage).



If your API is REST API — enable CORS per resource and deploy

1. API → left nav → **Resources** → click the resource (e.g., /getUploadUrl).
2. Actions → **Enable CORS**.
3. Set:
   - **Access-Control-Allow-Origin**: http://localhost:8000
   - **Allowed Methods**: POST,OPTIONS
   - **Allowed Headers**: Content-Type,Authorization
4. Confirm — console will create OPTIONS method and add method responses.
5. **Important**: Actions → **Deploy API** → choose stage dev → Deploy..

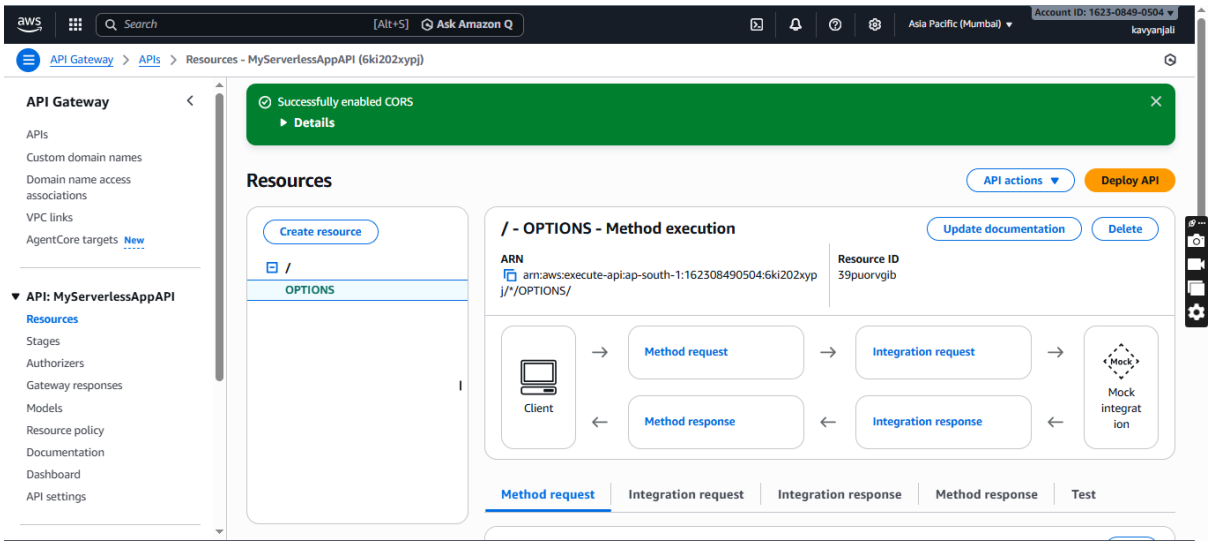


8) Enable CORS for your REST API methods

Even though your API is protected with Cognito, browsers will block requests unless CORS is set.

1. In your REST API console, left menu → **Resources**.
2. Select the resource you want to enable CORS on, e.g., /getUploadUrl.
3. With the resource selected, click **Actions → Enable CORS**.
4. Fill the popup exactly:
    - **Access-Control-Allow-Origin**:
    - http://localhost:8080
    - **Access-Control-Allow-Methods**:
    - POST, OPTIONS
    - **Access-Control-Allow-Headers**:
    - Content-Type, Authorization
    - **Enable credentials**: leave unchecked (unless you use cookies)
5. Click **Enable CORS and replace existing CORS headers**.

The console will automatically create an OPTIONS method for preflight requests and update your method responses.



9) Deploy the REST API

1. Left menu → **Actions → Deploy API**.
2. Choose your stage: dev (you created this in step 7).
3. Click **Deploy**.
4. After deployment, note the **Invoke URL** at the top of the stage page. Example:

https://<rest-api-id>.execute-api.ap-south-1.amazonaws.com/dev

All your methods now live under this base URL.
Example for your upload URL:

https://<rest-api-id>.execute-api.ap-south-1.amazonaws.com/dev/getUploadUrl

Summary & Conclusion

This project demonstrates the power and efficiency of **AWS serverless architecture** for building modern web applications. Key takeaways:

- **Scalability**: Using Lambda and API Gateway ensures the backend scales automatically based on user activity.
- **Security**: Cognito provides secure authentication, and S3 pre-signed URLs allow safe file uploads without exposing storage credentials.
- **Simplicity & Cost Efficiency**: No server management is needed, reducing operational overhead and cost.
- **Modularity**: Each service (Cognito, S3, DynamoDB, Lambda) handles a specific responsibility, making the system modular and maintainable.

Conclusion:
This serverless web application project is a practical example of building a secure, scalable, and cost-effective cloud-native solution using AWS managed services. It provides hands-on experience in **API integration, authentication, file management, and serverless computing**, which are essential skills for modern cloud developers and architects.