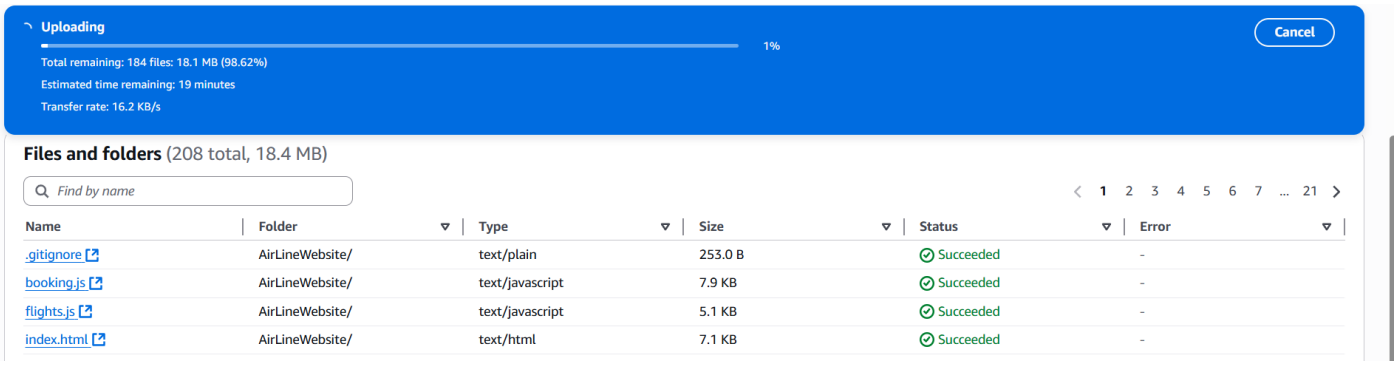
✈ Airline Ticket Booking System – AWS Cloud Project

AWS Services You Will Use

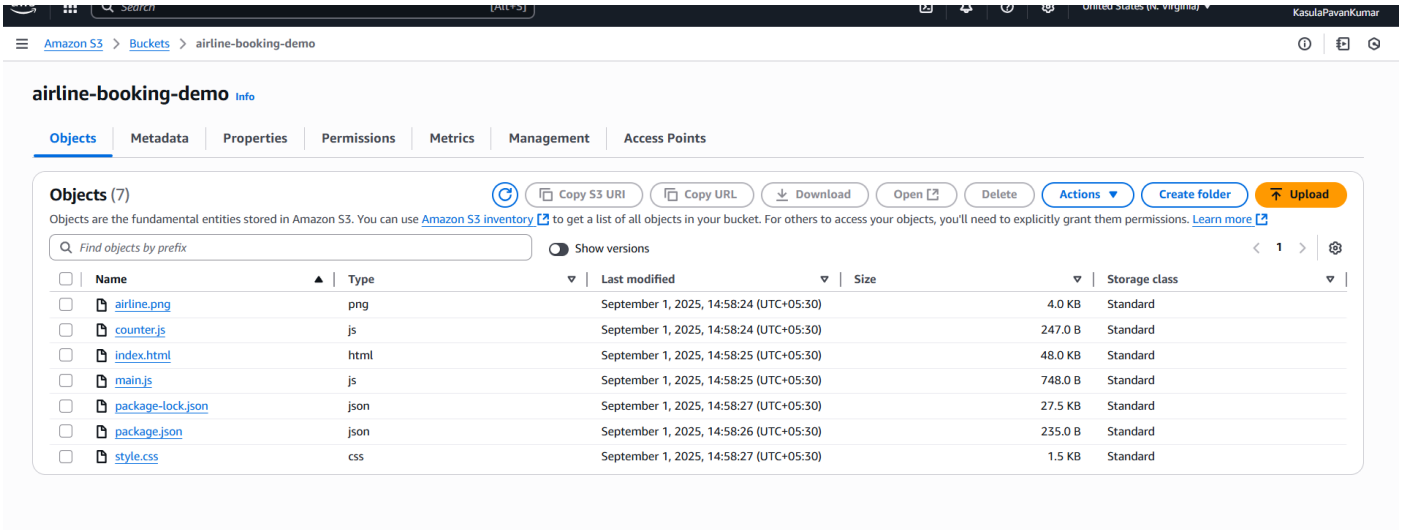
- **Amazon S3** → Host the website (UI)
- **Amazon API Gateway** → Expose APIs
- **AWS Lambda** → Business logic (search & booking)
- **Amazon DynamoDB** → Store flight & booking info
- **Amazon SES** → Send confirmation emails
- **IAM** → Permissions & security
- **SES** – Email Service
- **CloudWatch** – log Groups

➔ **Step 1: Setup Flight Booking Website**

- Create an S3 Bucket
- Enable Static Website Hosting
- Upload Front-End Files
- Make Files Public
- Test Website



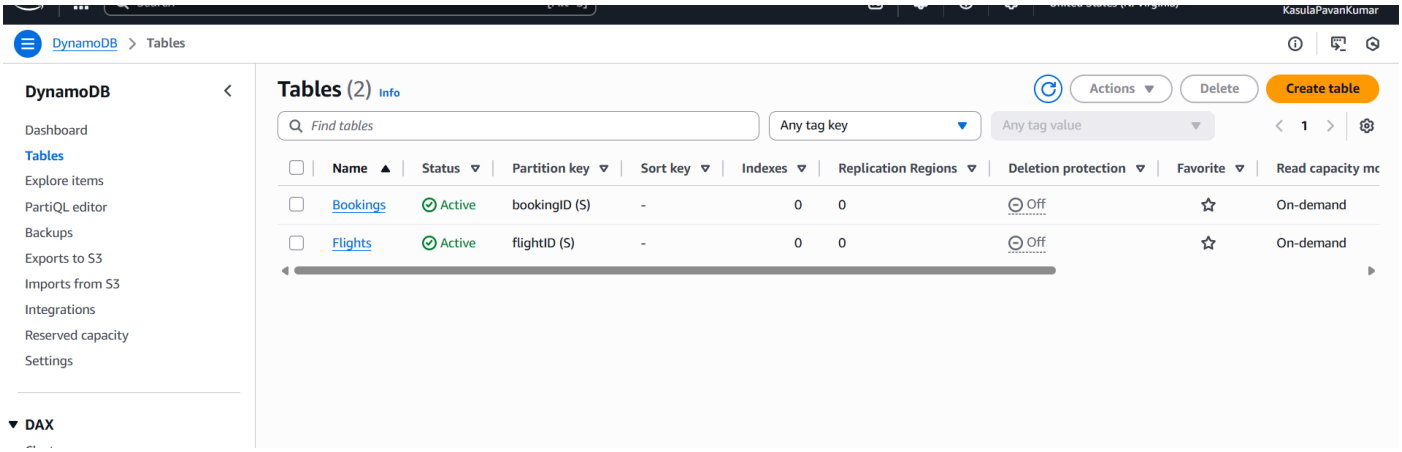
S3 creation Image(1a)



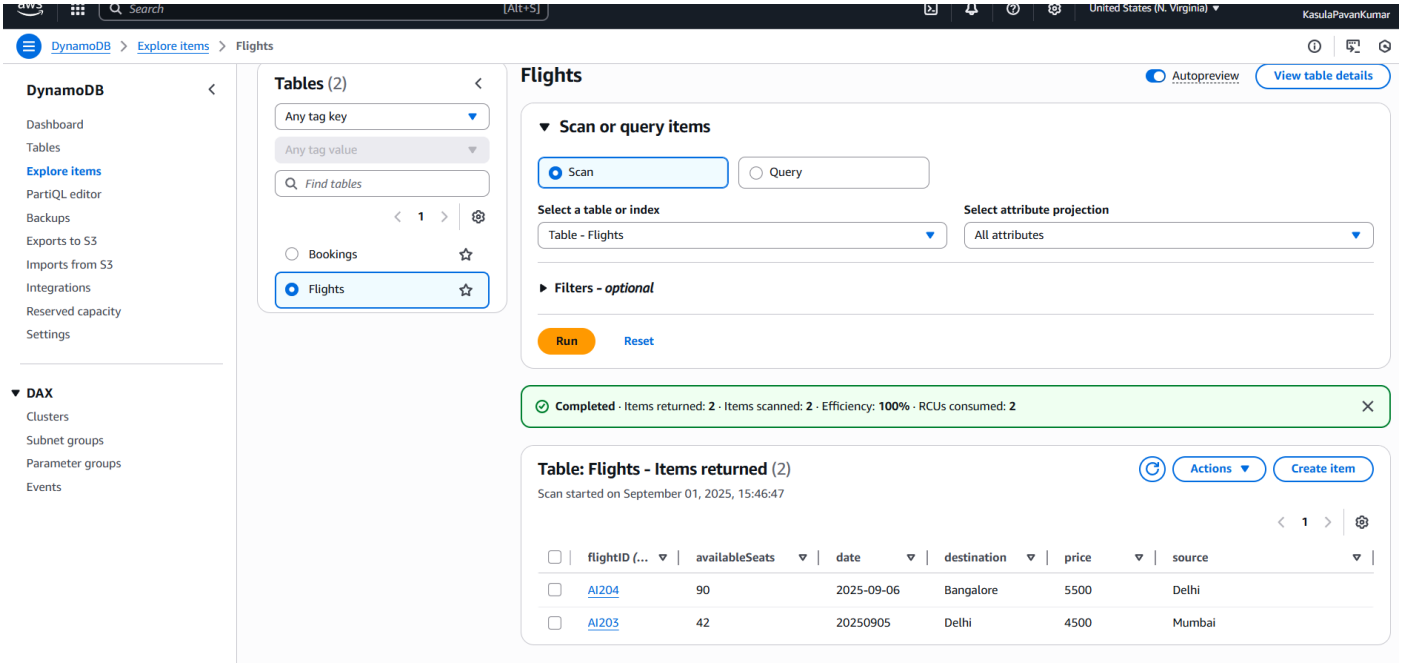
S3 creation Image(1b)

➔ Step 2: Create DynamoDB Tables

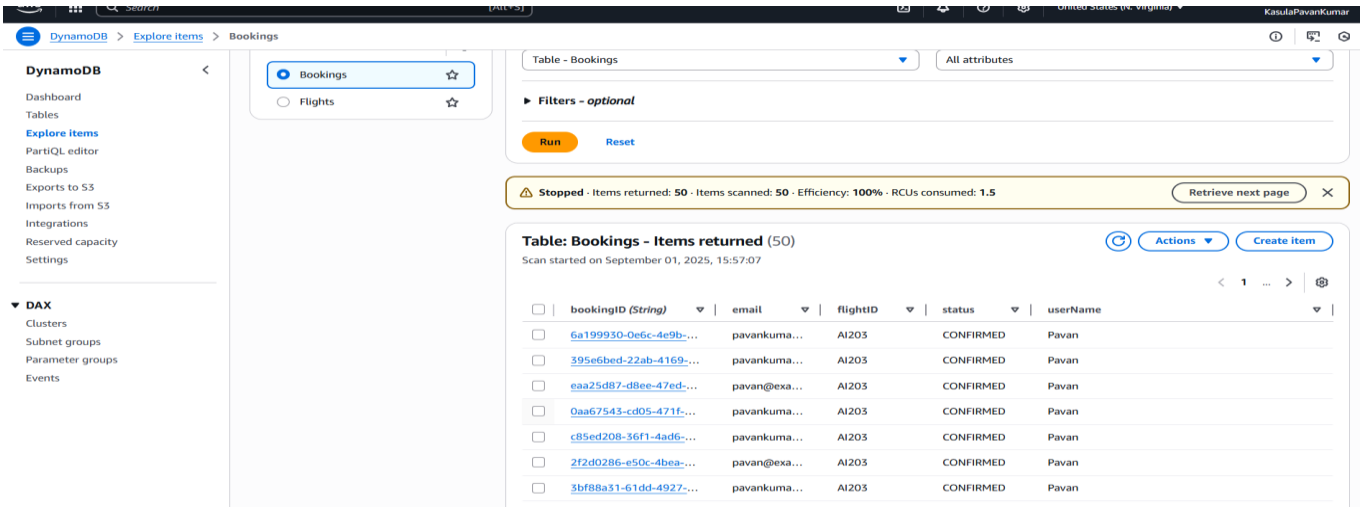
- Create Flights Table
- Add Sample Flight Data (after table is created)
- Create Bookings Table



DynamoDB creation Image(1a)



DynamoDB creation Image(1b)



DynamoDB creation Image(1c)

➔ **Step 3: Create Lambda Functions**

We need two **serverless backend functions**:

- searchFlights → Fetch available flights from DynamoDB.
- bookTicket → Store booking info + reduce available seats + send confirmation later.

Create Lambda Function – searchFlights(Permissions → Create a new role with **DynamoDB read-only acess**

Create Lambda Function – bookTicket(Permissions → Role with **DynamoDB full access + SES)**

Runtime → **Python 3.12** (or Our Wish Based on Knowledge about project -SP).

Book Ticket code (Lambda) **–**

```
import json
import boto3
import uuid

dynamodb = boto3.resource('dynamodb')
flights_table = dynamodb.Table('Flights')
bookings_table = dynamodb.Table('Bookings')
ses = boto3.client('ses')

def lambda_handler(event, context):
    try:
        # Parse body from API Gateway
        flight_id = event.get("flightID")
        user_name = event.get("userName")
        email = event.get("email")

        if not flight_id or not user_name or not email:
            return {
                "statusCode": 400,
                "headers": {"Access-Control-Allow-Origin": "*"},
                "body": json.dumps({"message": "Missing required fields"})
            }
```

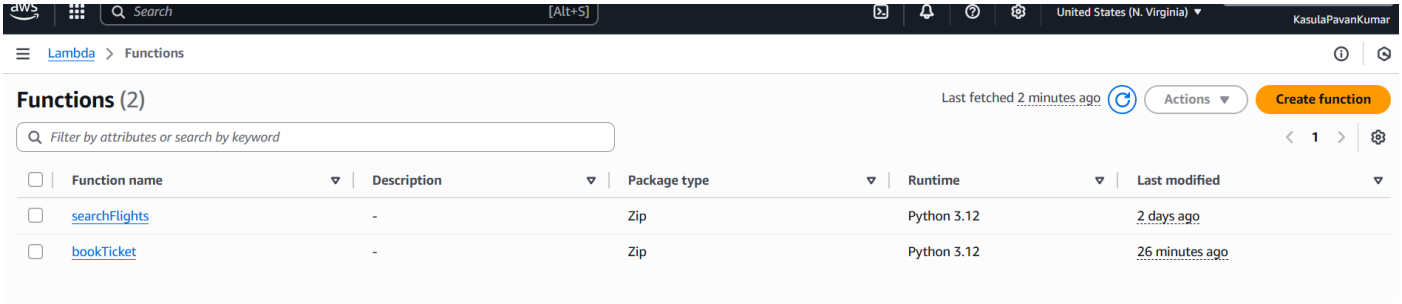
```
# Generate booking ID
booking_id = str(uuid.uuid4())

# Save booking
bookings_table.put_item(
    Item={
        "bookingID": booking_id,
        "flightID": flight_id,
        "userName": user_name,
        "email": email,
        "status": "CONFIRMED"
    }
)

# Decrease available seats (ensure not below zero)
flights_table.update_item(
    Key={"flightID": flight_id},
    UpdateExpression="SET availableSeats = if_not_exists(availableSeats, :start) - :val",
    ExpressionAttributeValues={":val": 1, ":start": 100},  # default 100 seats if missing
    ConditionExpression="availableSeats >= :val"
)

# Send confirmation email
ses.send_email(
    Source="pavankumarkasula73@gmail.com",  # must be verified in SES
    Destination={"ToAddresses": [email]},
    Message={
        "Subject": {"Data": "Flight Booking Confirmation"},
        "Body": {
            "Text": {
```

```
                "Data": f"Hello {user_name},\n\nYour booking for flight {flight_id} is CONFIRMED.\nBooking ID: {booking_id}\n\nThank you for choosing our Airline!"
            }
        }
    }
)

    return {
        "statusCode": 200,
        "headers": {"Access-Control-Allow-Origin": "*"},
        "body": json.dumps({
            "message": "Booking confirmed & email sent",
            "bookingID": booking_id
        })
    }

  except Exception as e:
    print("Error:", str(e))
    return {
        "statusCode": 500,
        "headers": {"Access-Control-Allow-Origin": "*"},
        "body": json.dumps({"message": "Internal server error", "error": str(e)})
    }
```

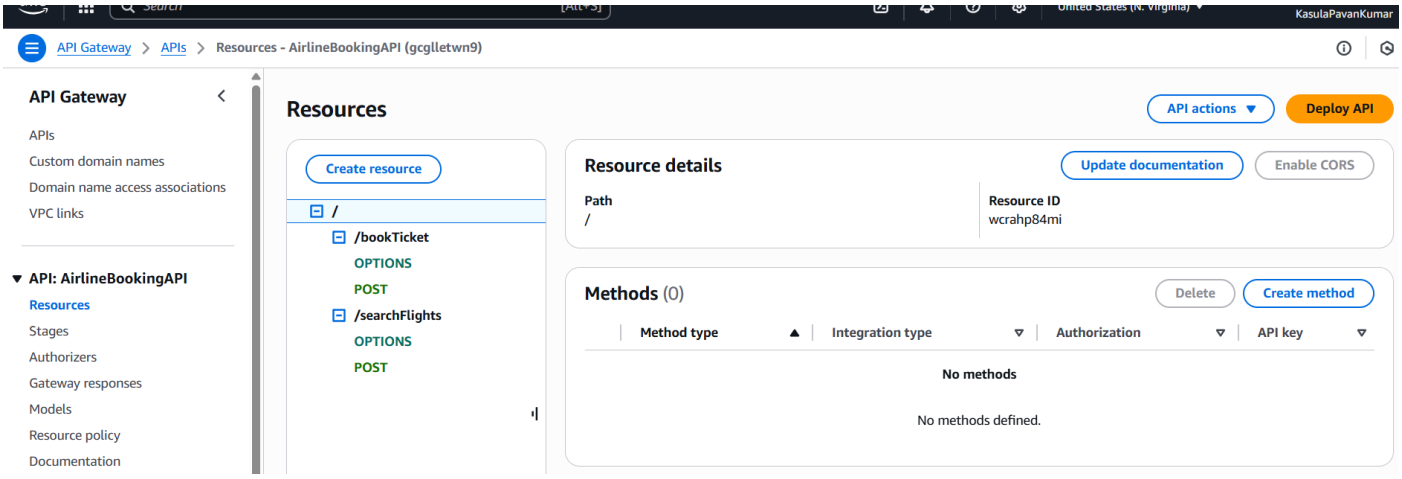
Search Flights Code (Lambda) :

```
import json

def lambda_handler(event, context):
  # Get the 'body' field
  body = event.get('body', '{}')
```

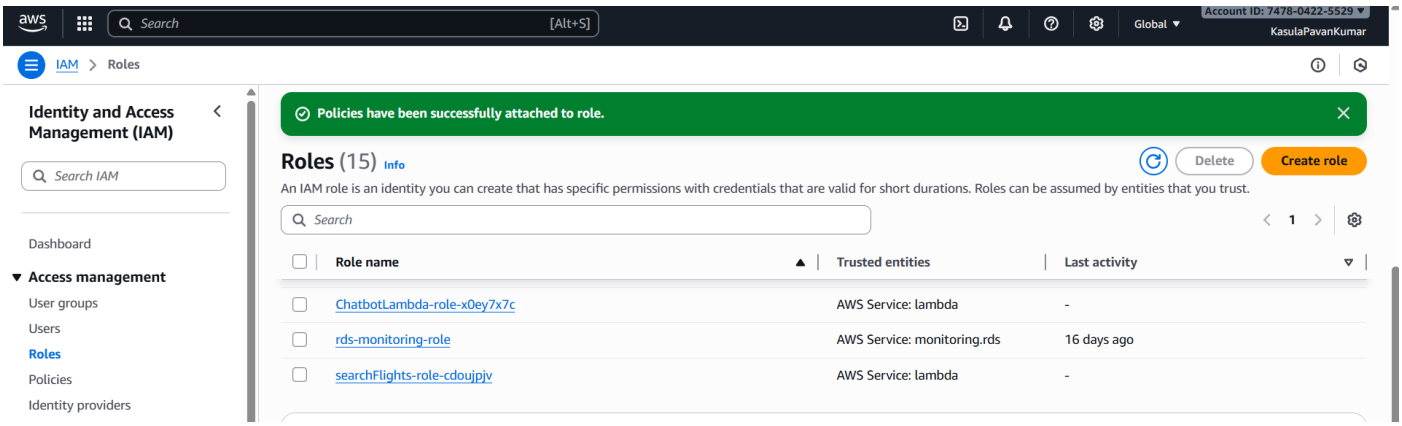
```
# If 'body' is already a dict, use it directly.
# If it's a string, parse it to dict.
if isinstance(body, dict):
    data = body
else:
    data = json.loads(body)

# Now you can safely access keys
source = data.get('source')
destination = data.get('destination')
date = data.get('date')

# Your logic here...
return {
    'statusCode': 200,
    'body': json.dumps({
        'source': source,
        'destination': destination,
        'date': date
    })
}
```



Lambda function image(1a)



Lambda function image(1b)



Lambda function image(1c)

Step 4: Create APIs with Amazon API Gateway

- Here we connect your frontend (S3 website) to the backend (Lambda functions).
- **Open API Gateway ->** Choose **REST API**
- Create Resource for Flights Search -> selected → Click Actions → Create Method → Choose POST
- Create Resource for Booking Tickets -> selected → Click Actions → Create Method → Choose POST
- Enable CORS (Frontend Access)
- Finally we Deploy our API here.(copy & paste your api endpoints in notepad for future use case.
- After we test API'S

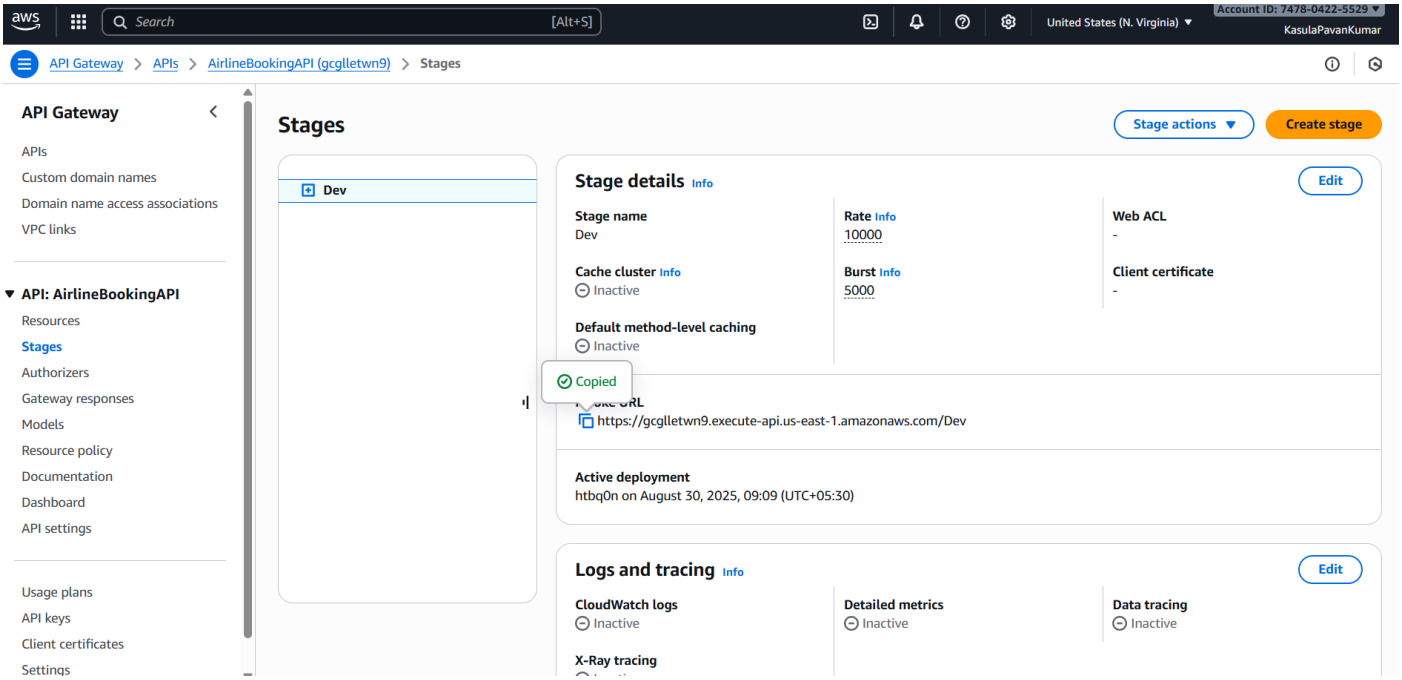
  Search Flights Api test code –

```
{
 "source": "Mumbai",
 "destination": "Delhi",
"date": "2025-09-05"
}
```

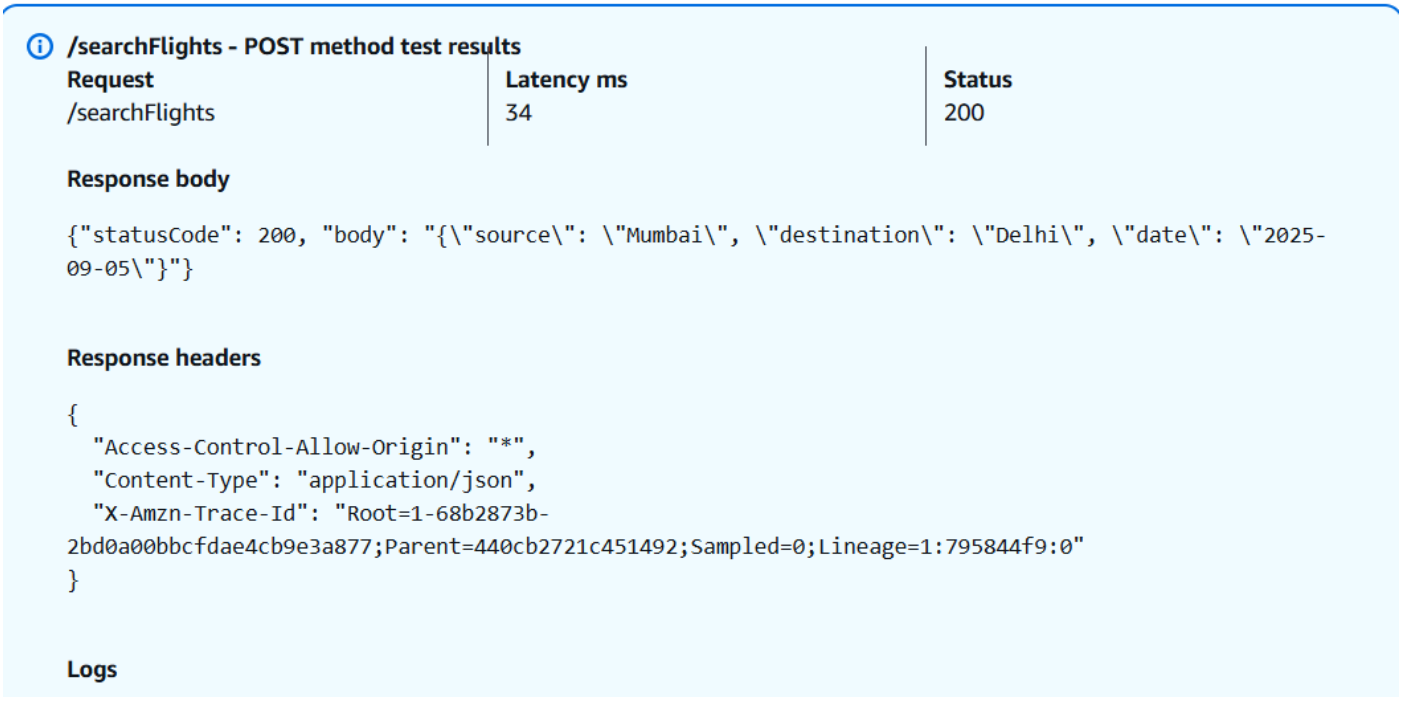
- After we test API’S

  Ticket Booking Api test code –

```
{
"flightID": "AI203",
"userName": "Pavan",
"email": "pavan@example.com"
}
```



Api Gateway Image(1a)



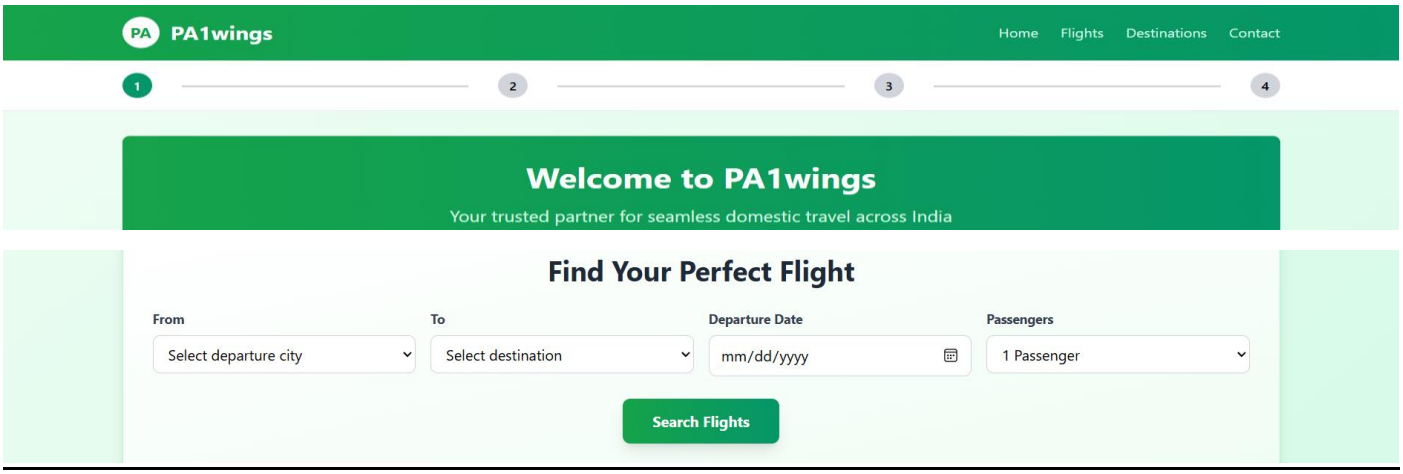
Api Gateway Testing(post) Image(1a)

➔ **Step 5(a): Connect Frontend with APIs.**

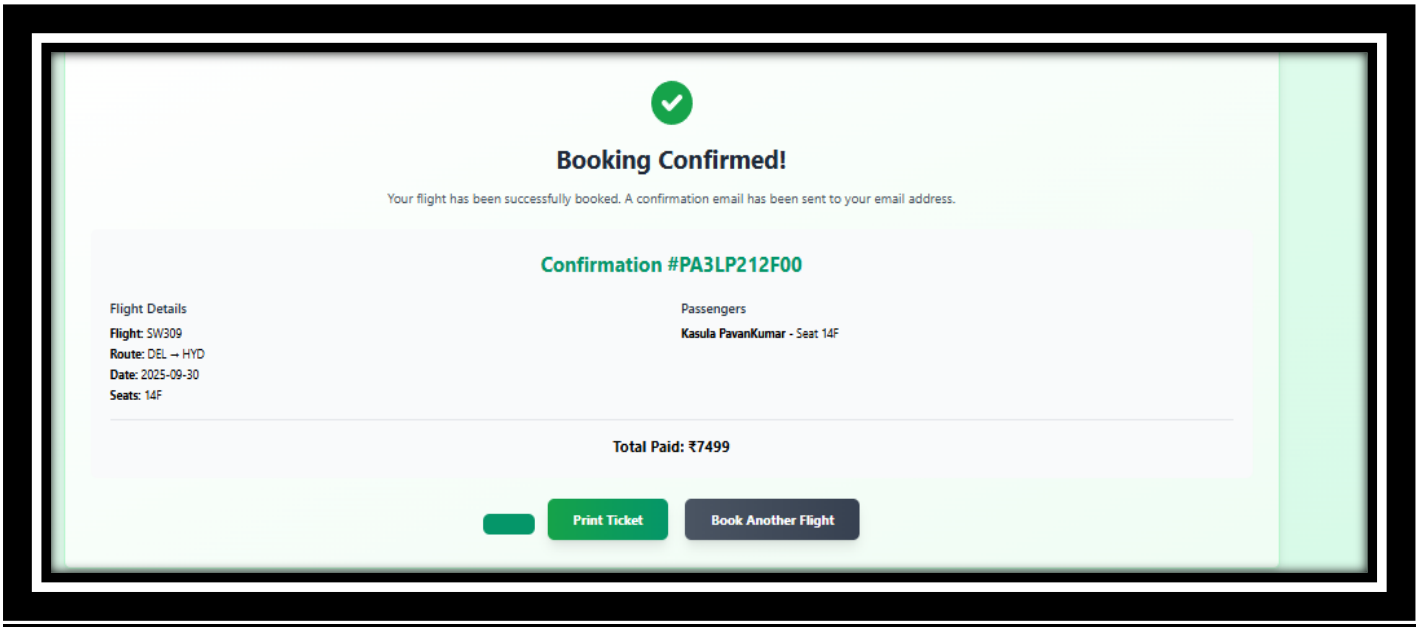
- Get API Gateway URL
- We'll use this in frontend code for connection between them
- Update URL in required fields in frontend.
- Upload Updated Files to S3
- Refresh our website URL in S3 and check.
- After Booking Ticket Booking we need to get booking details to used Gmail.

➔ **Step 5(b): Email Confirmation with Amazon SES.**

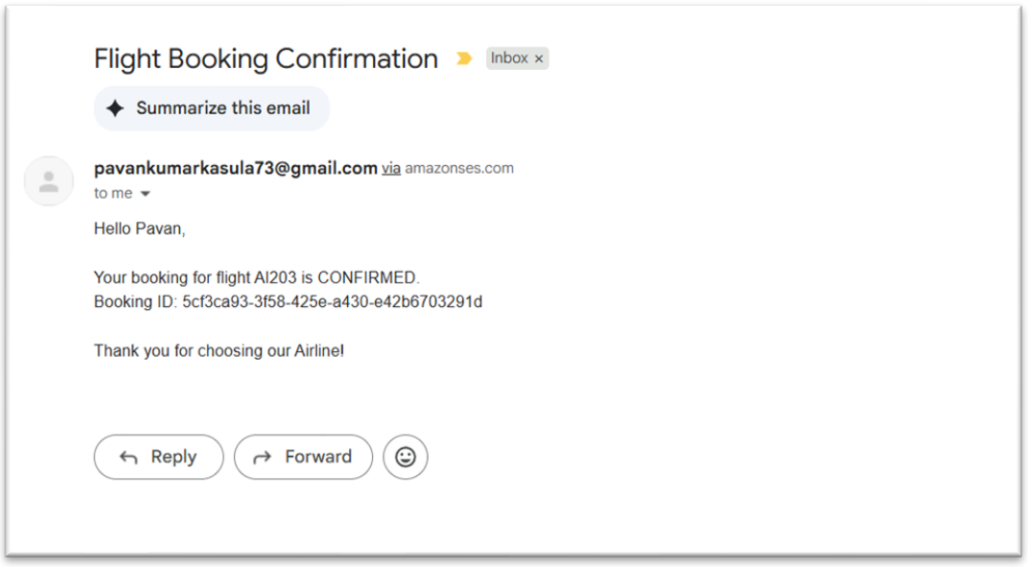
- In Verified identities → Create identity
- Choose **Email address**.
- Enter your email (e.g., pavankumarkasula73@gmail.com).
- You'll get a verification email → Click the link to confirm.
- **Add SES Permission to Lambda -> Your bookTicket Lambda role**
- Attach policy → AmazonSESFullAccess



Hosted Website(S3/EC2) Image(1a)

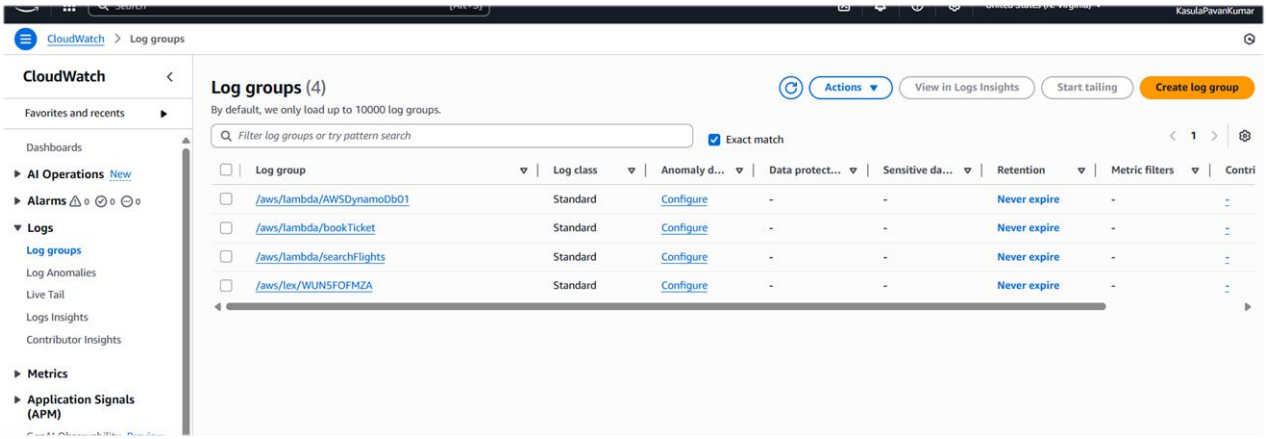


Hosted Website (Booking Completed) Image(1b)

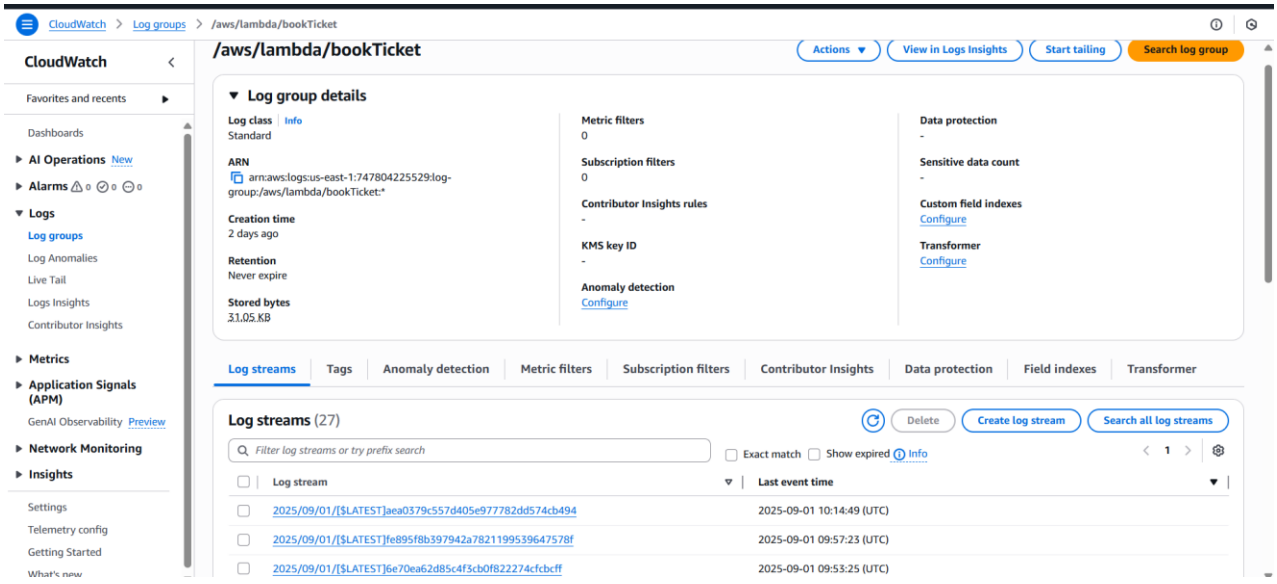


Hosted Website (MAIL CONFIRMATION) Image(1c)

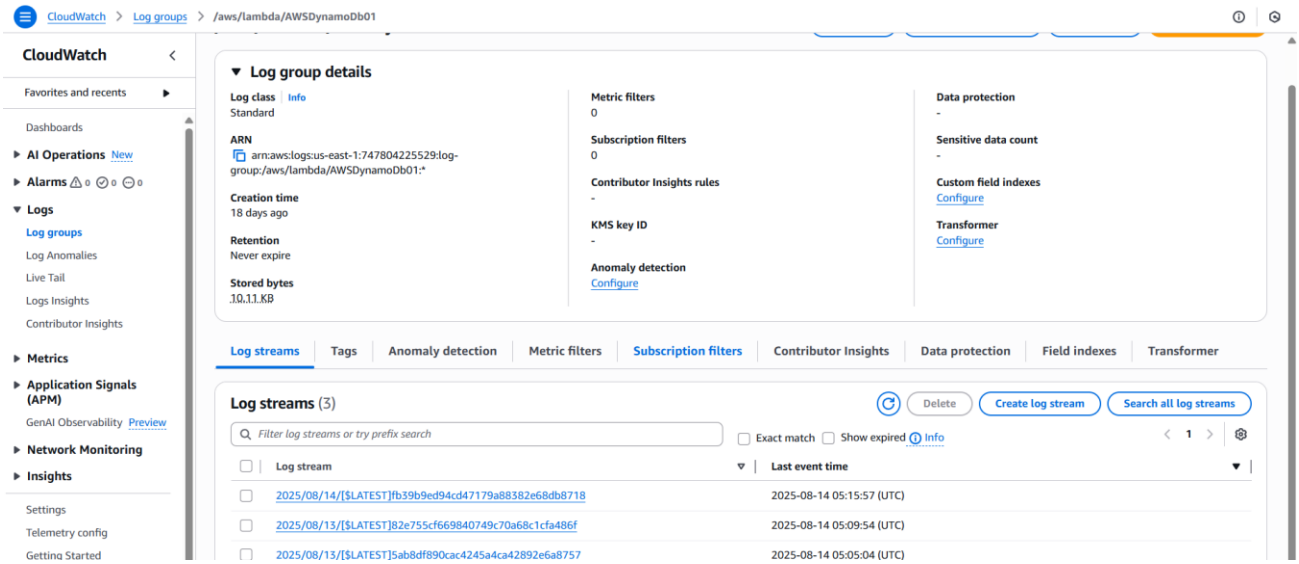
Check CloudWatch Log metrics for Log analysis for Errors.



Cloudwatch(Logs)(1a)



Cloudwatch(Logs)(1b)



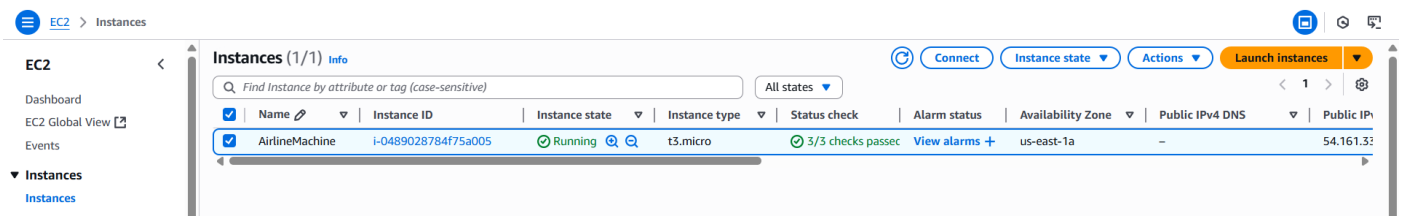
Cloudwatch(Logs)(1c)

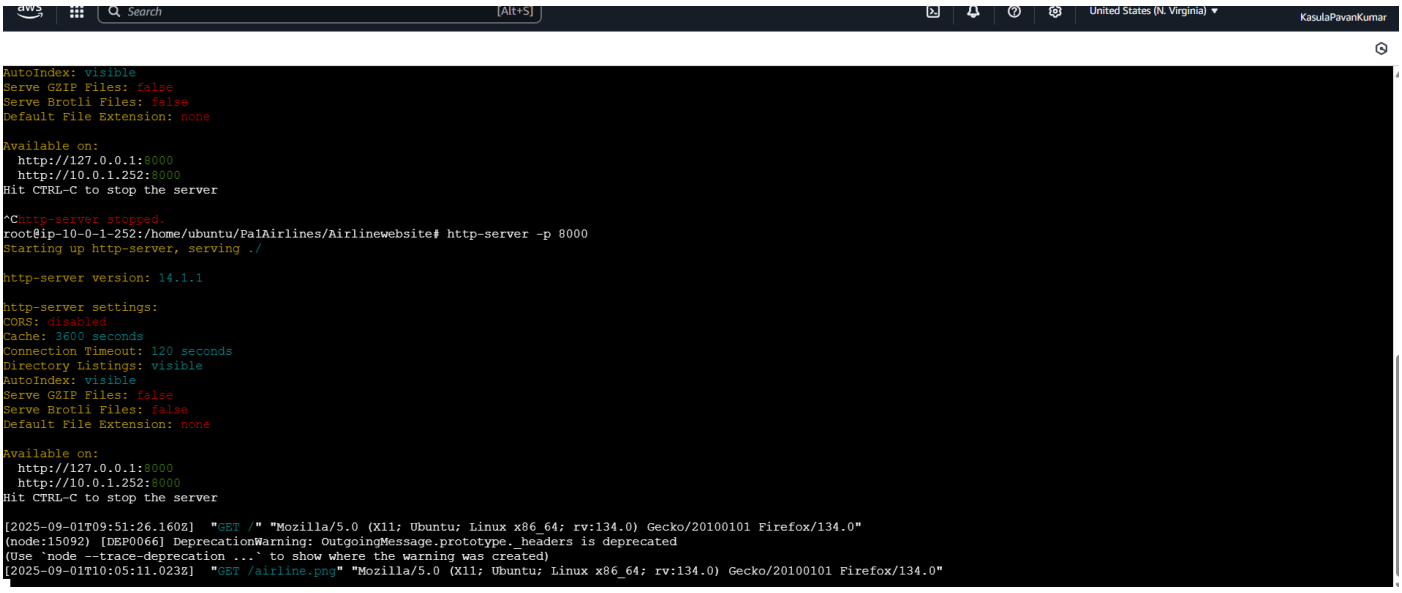
➔ STEP – 6(OPTIONAL)

In case of trouble with S3 bucket static website, we can create an EC2 machine and clone the repo there, and install required packages in machine , After check with machine ip and 8000.

EC2 MACHINE : CONTAINS WEBSITE(UPDATED VIA GIT CLONE)

http-server -p 8000





➔ **LEARNINGS & CHALLENGES FROM THESE PROJECT:**

- Gained practical experience with S3 (static website hosting), DynamoDB (NoSQL database), API Gateway (REST API management), Lambda (serverless functions), and SES (email notifications).
- Learned how these services interact to form a complete cloud-based application.
- Understood how to build applications without managing servers using AWS Lambda.
- Learned the importance of IAM roles & permissions to securely allow services to talk to each other.
- Learned how browser security policies (CORS) affect API calls and how to configure API Gateway headers to make frontend and backend communicate smoothly.
- Explored DynamoDB partition keys and schema design for handling **Flights** and **Bookings** efficiently.
- Integrated SES to send booking confirmation emails, which introduced me to **event-driven notifications**.

➔ **CHALLENGES:**

- Initially, My Lambda function failed because request payloads come wrapped inside event["body"]
- The frontend fetch () calls were blocked due to missing CORS headers.
- Fixed this by enabling CORS in API Gateway and adding headers in Lambda responses.
- Logs in CloudWatch and API Gateway execution logs were critical to identify why Lambda was returning 400 Bad Request.
- After booking a ticket, I noticed that no confirmation email was received.
  This created a lot of confusion since the Lambda executed successfully, but the user never got the expected notification.

  Guidance received from Sir / Madam:
  My guide suggested two possible troubleshooting paths:
- Deploy the frontend on **EC2** instead of S3 to rule out any static hosting restrictions.
- Double-check whether the **API Gateway invoke URL** was correctly configured in the frontend fetch () code.
- Learned to trace problems step by step using logs and by testing services independently (e.g., calling API Gateway directly via Postman).
- Gained awareness that debugging in cloud projects isn't just about fixing code—it also requires validating **infrastructure setup and integrations**.
- Learned to rely on **logs, monitoring, and guided troubleshooting** rather than guessing.
- Understood that sometimes **workarounds** (like deploying on EC2) help isolate issues, even if the root cause is elsewhere.