# Enterprise Cloud and Distribution Web Application
# Individual Assignment - 02


**Mohammed Mawsoon Maryam Mashkoora CB010658_21035976**

**BSc (Hons) Computer Science**

**Level 06**

**Lecturer**: Mr. Nisala Aloka Bandara

**Batch Code:** HFK2422COMF

# Acknowledgment

With a great appreciation, I would like to thank Mr. Nisala Aloka Bandara for all of the help, advice, and knowledge during this assignment. Their opinions and thoughts have greatly influenced the direction of this work.

Finally, I would like to thank my family and friends for their unwavering support and encouragement throughout this academic endeavor. I also want to express my gratitude to my coworkers and classmates for their supportive conversations and insightful debates that have improved my comprehension of the subject. Lastly, I want to express my gratitude to my friends and family for their continuous support and motivation during this academic journey.

CB010658

# Contents



# Enterprise Cloud and Distribution Web Application

# 1. Introduction

The DreamStreamer is an innovative AI-driven platform that designs music recommendations, considering the tastes and preferences of a user, with the commitment to raise the bar for users' listening experiences. This report describes the realization of a proof-of-concept website for DreamStreamer, illustrating its offerings and making user engagement dynamic.

The aim of this project is to design and implement a robust, scalable web application that would be hosted on AWS, utilizing a host of cloud services that meet the operational requirements of DreamStreamer. An admin dashboard will also be provided with the website to manage information about music and artists efficiently, thereby allowing the administrators to create, read, update, and delete data. Customers will be able to explore and filter music albums by genres, artists, and other parameters using a user-friendly interface that the system will offer. The website, hosting static material on AWS S3, serverless backend operations on AWS Lambda, and a suitable database solution capable of handling high-throughput queries and analytics will all enable the aforementioned functionalities.

This architecture provides full reliability and performance, thus setting a very solid base for further enhancements in line with platform evolution.

This will be followed by the technical implementation, the technology stack used, and optimization strategies to provide a better user experience without compromising on smooth interaction with the infrastructural systems that support the service.

# 2. Solution Architecture

This architecture reaps all the advantages of various AWS components used in it, which are S3, Lambda, API Gateway, DynamoDB, and RDS, towards providing efficient and cost-effective solutions on both customer-facing and admin interfaces.

## 2.1 Overview of the architecture

| AWS Component | Used |
|---|---|
| Amazon S3 | It used to host the website's static assets such as HTML, CSS, JavaScript, and album art images and the song track |
| AWS Lambda | To do all the CRUD functionality for admin to the created Album and Songs database |
| Amazon API Gateway | To interconnect front end and back end by triggering the lambda function. |
| Amazon DynamoDB | To store Song and Album details to do the CRUD |
| Amazon RDS | RDS is used to store the analytics of the popular albums. |
| Amazon SNS | Used to send notifications, such as inventory reports, to the admin via email |
| AWS IAM | IAM is used to securely manage access to AWS resources which here created as admin |
| AWS CloudFront | Used to optimize the website performance. |
| AWS Cognito | Uset to give the authentication action for admin and the normal users. |

## 2.2 AWS Solution Diagram

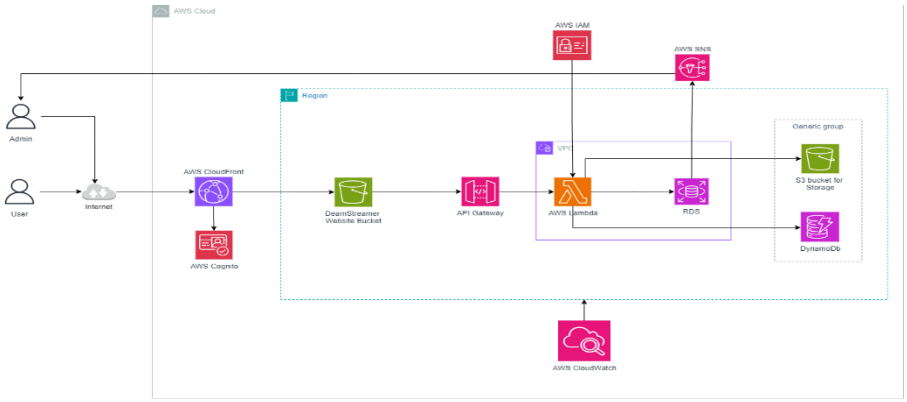



*Figure 1.Customer and Admin AWS Architecture*

## 2.3. Implementation of Solution Diagram For Services

### 01)Amazon S3

AWS S3 provides scalability, reliability, and cost-effectiveness that make it ideal for hosting static websites. In this solution Album art images and a song tracks, which safely stored in different s3 bucket and delivered through it with high availability and ease of access for the users.

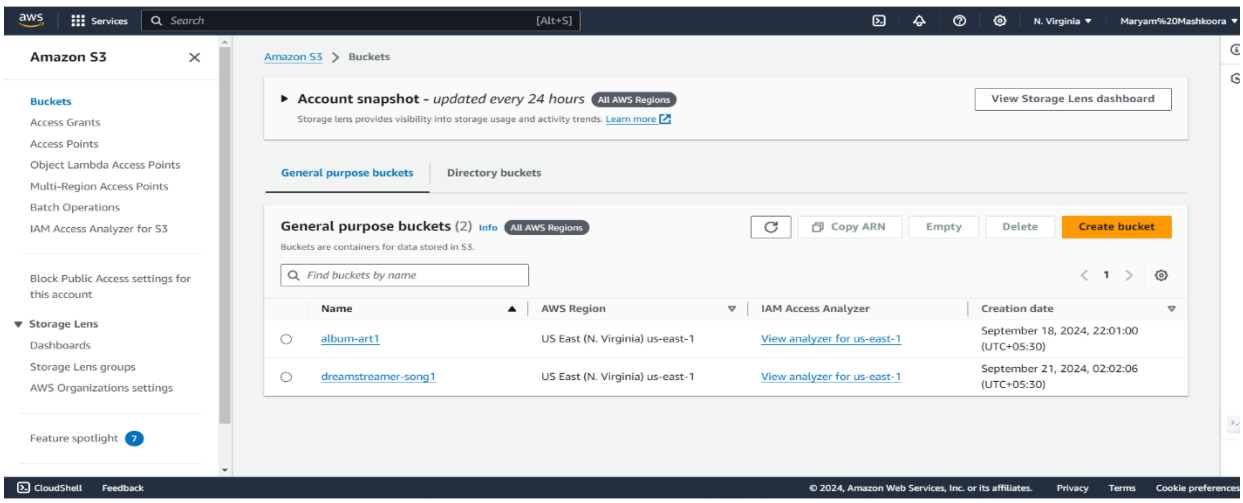



*Figure 2.S3 Bucket*

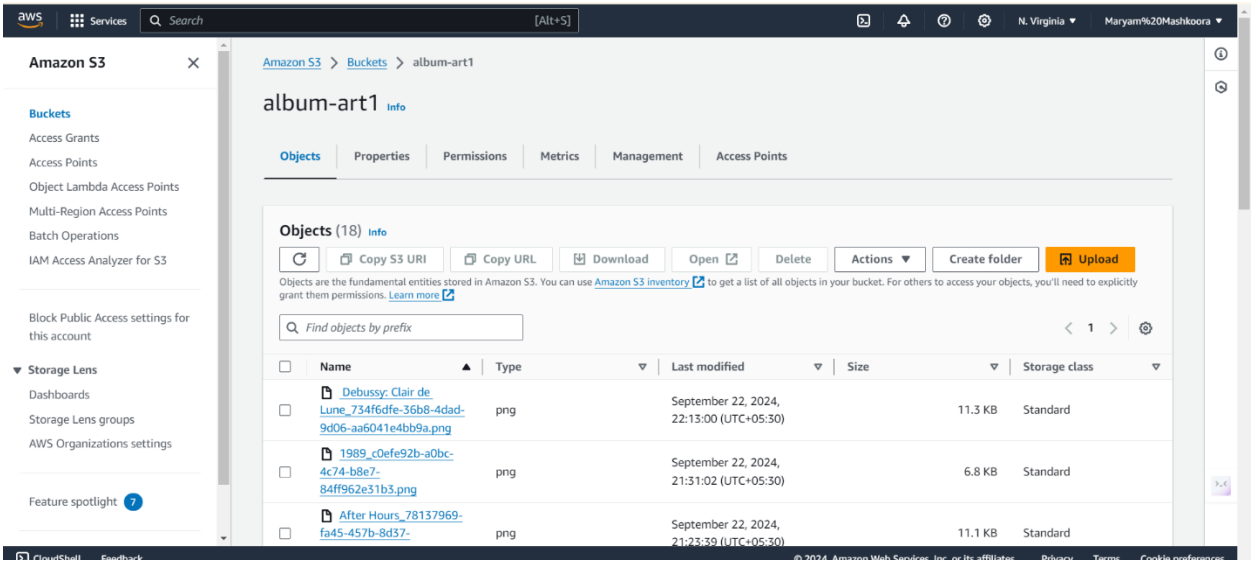



*Figure 3. Collection on Album art images in S3 bucket*

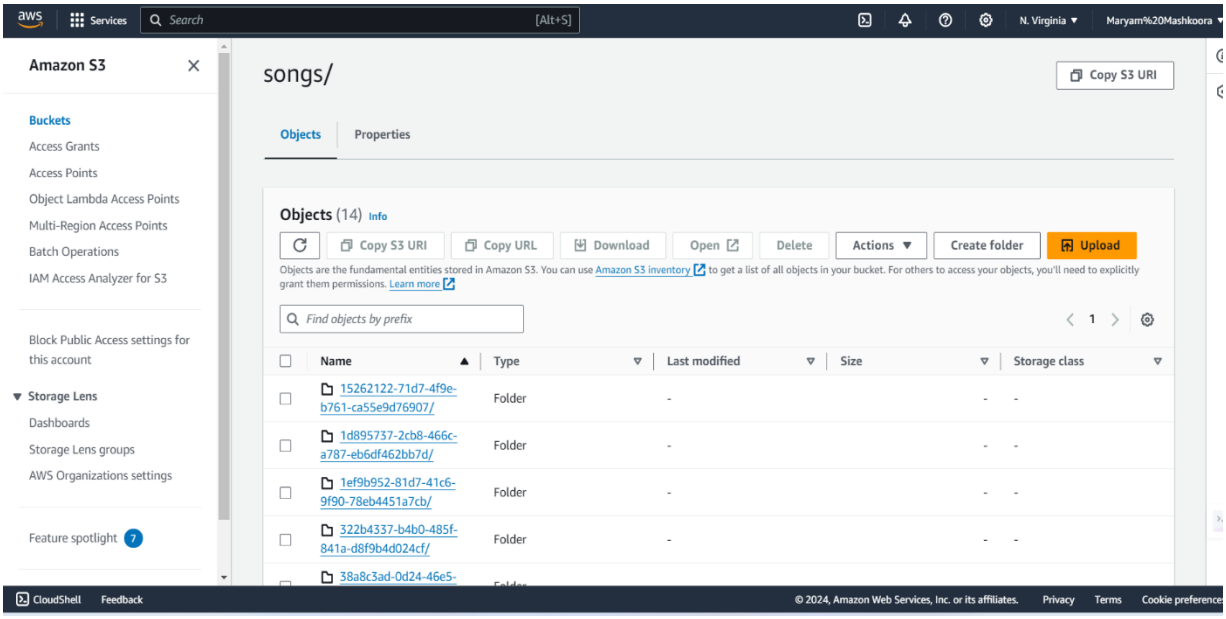



*Figure 4.Collection of song audio files stored S3 Bucket*

02) AWS Lambda

AWS Lambda is used because it introduces the application to a serverless architecture that automatically scales, without the need to manage servers. Lambda can handle CRUD operations to the database and process API requests quite efficiently, making it great for cost-effective, event-driven applications. Like here, I have used two different CRUD one for Album and other is Songs where admin can do CRUD operation. And also this system also create a lambda function for the analytics to retrieve the data for the RDS and to do the SNS service.

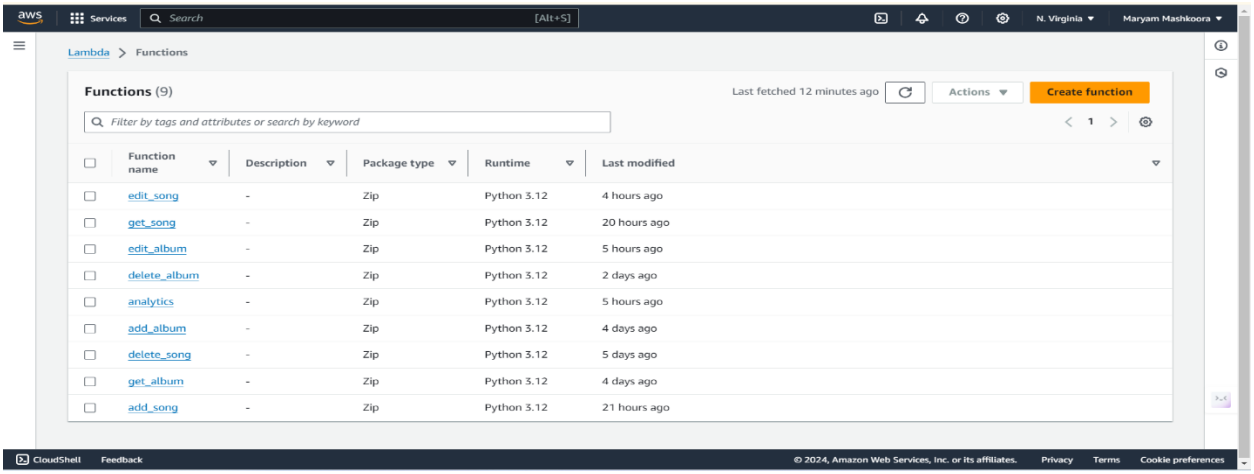



*Figure 5.AWS Lambda Function*

## 03) AWS API Gateway

This structure use API gateway to create RESTful APIs for securely connect with the backend. Here, It is created different resources and each resources are provided with methods and while creating the each method the appropriate lambda function which is created is integrated.

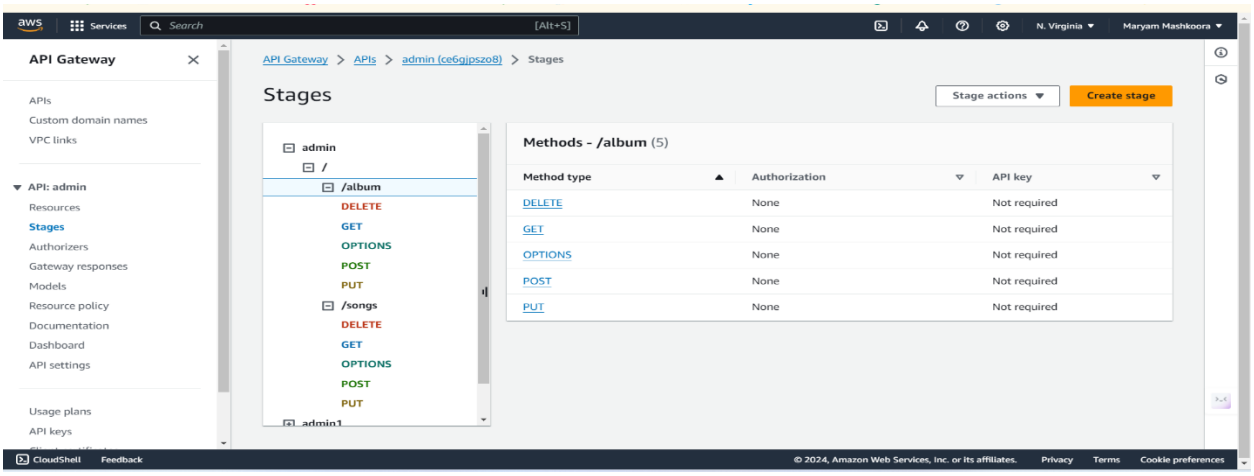
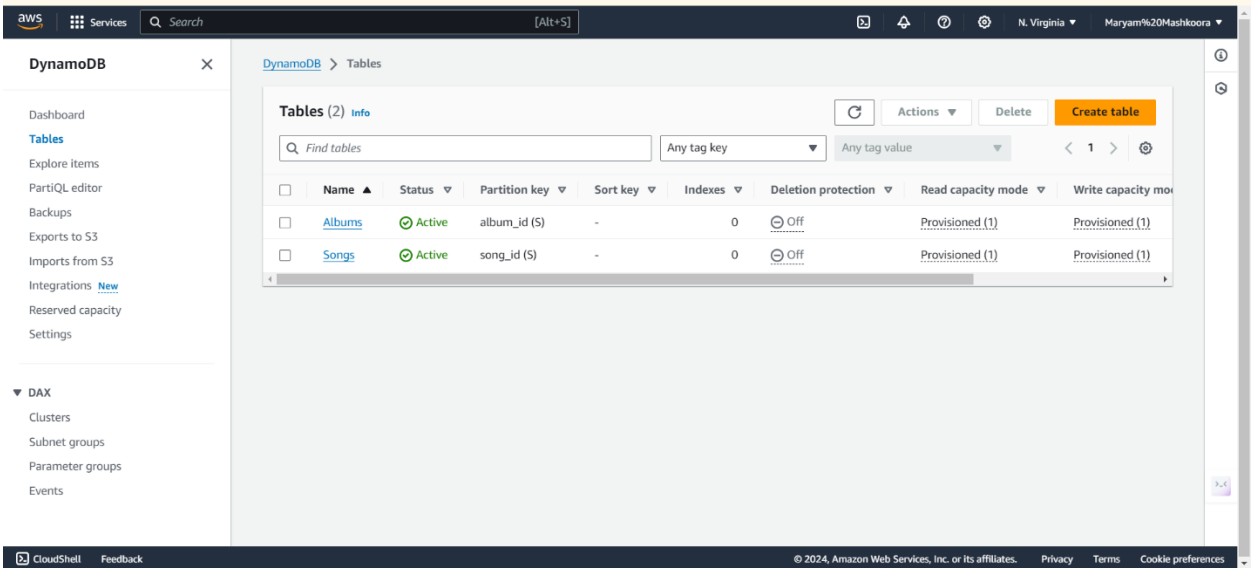


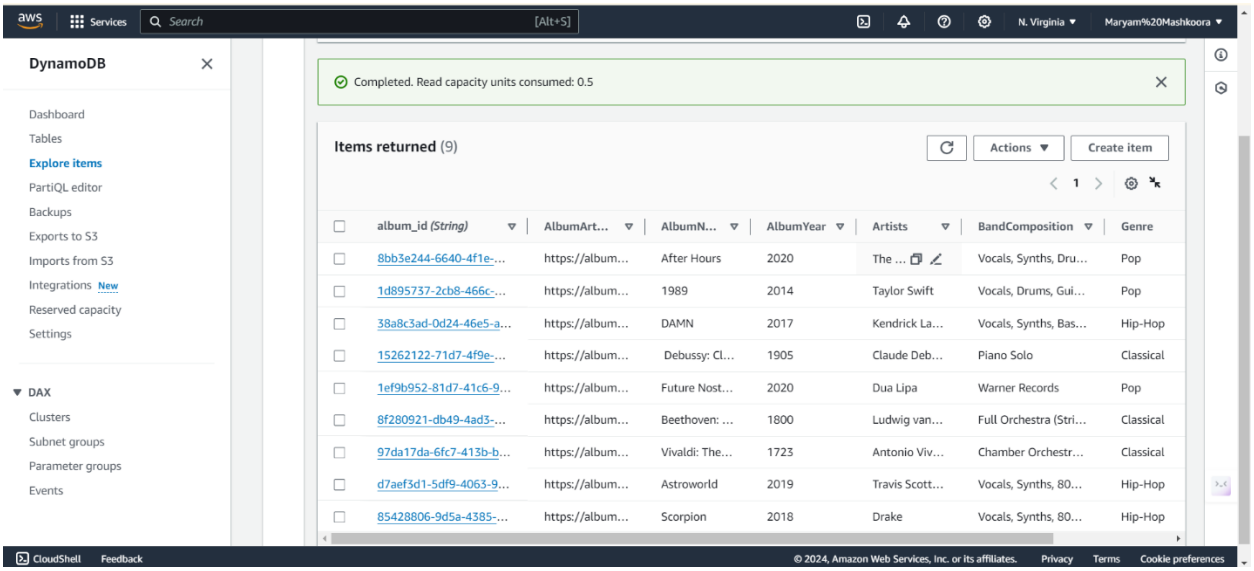

*Figure 6.AWS API Gateway*

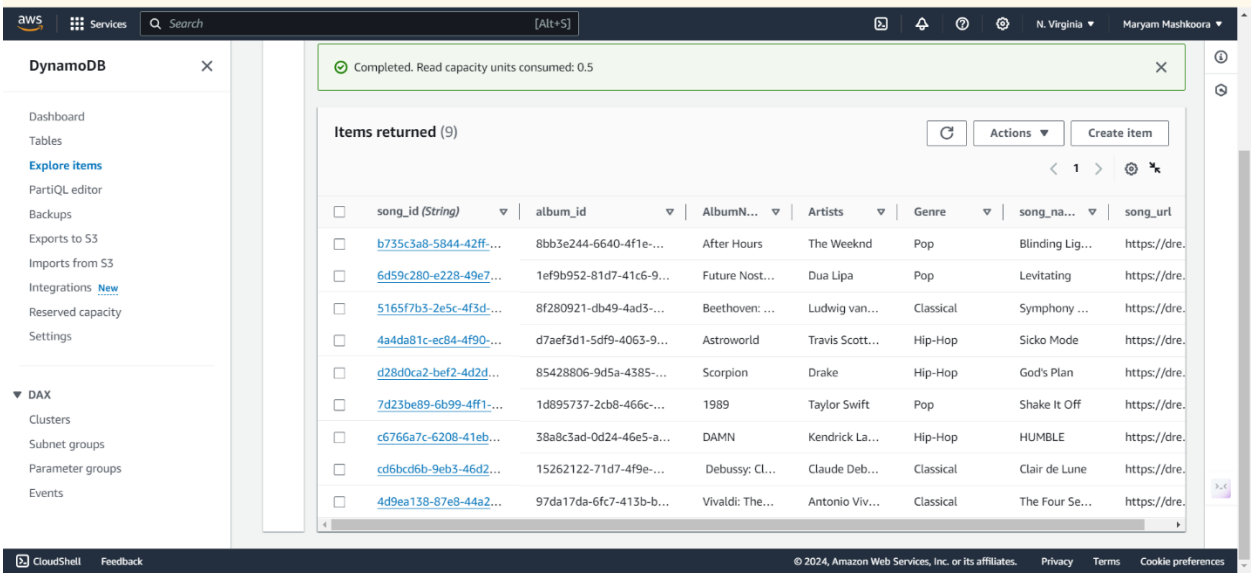
## 04) AWS DynomoDb

For this scenario I have created two databases as “Albums” and other “Songs” where, album_id will be a reference key of the song table where admin can add songs for available albums. This database is used for the CRUD operation and also for the filtration process in both Admin dashboard and User Interface.

CB010658

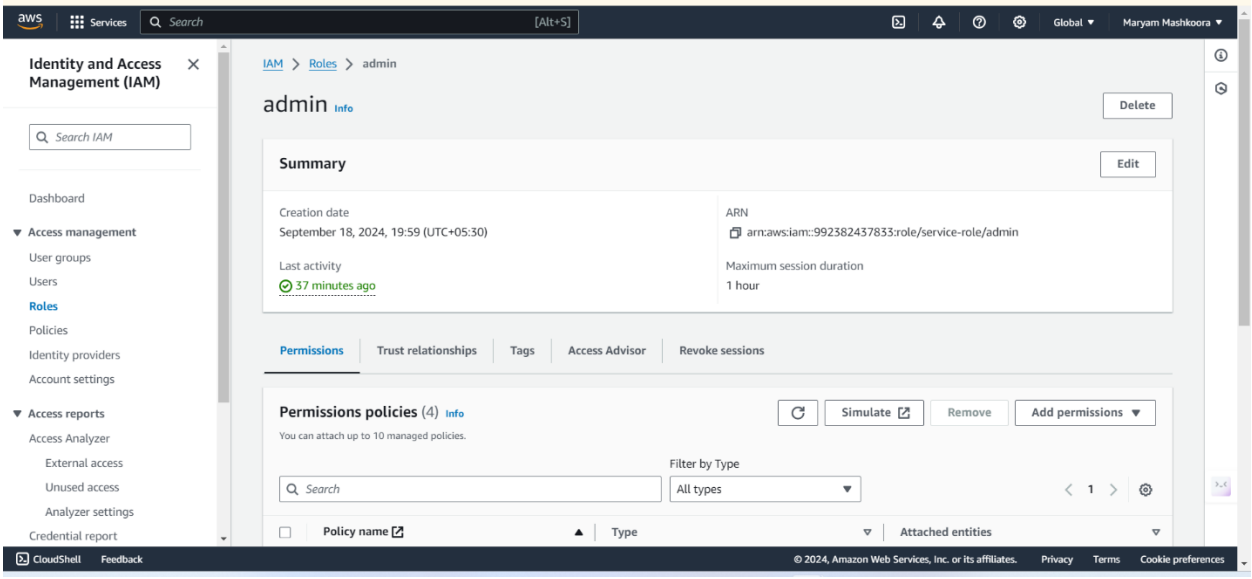

aws
Services
Search
[Alt+S]
N. Virginia
Maryam%20Mashkoora
DynamoDB
Dashboard
Tables
Explore items
PartiQL editor
Backups
Exports to S3
Imports from S3
Integrations New
Reserved capacity
Settings
DAX
Clusters
Subnet groups
Parameter groups
Events
DynamoDB > Tables
Tables (2) Info
Actions
Delete
Create table
Find tables
Any tag key
Any tag value
Name
Status
Partition key
Sort key
Indexes
Deletion protection
Read capacity mode
Write capacity mo
Albums
Active
album_id (S)
0
Off
Provisioned (1)
Songs
Active
song_id (S)
0
Off
Provisioned (1)
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


aws
Services
Search
[Alt+S]
N. Virginia
Maryam%20Mashkoora
DynamoDB
Dashboard
Tables
Explore items
PartiQL editor
Backups
Exports to S3
Imports from S3
Integrations New
Reserved capacity
Settings
DAX
Clusters
Subnet groups
Parameter groups
Events
Completed. Read capacity units consumed: 0.5
Items returned (9)
Actions
Create item
album_id (String)
AlbumArt...
AlbumN...
AlbumYear
Artists
BandComposition
Genre
8bb3e244-6640-4f1e-...
https://album...
After Hours
2020
The ...
Vocals, Synths, Dru...
Pop
1d895737-2cb8-466c-...
https://album...
1989
2014
Taylor Swift
Vocals, Drums, Gui...
Pop
38a8c3ad-0d24-46e5-a...
https://album...
DAMN
2017
Kendrick La...
Vocals, Synths, Bas...
Hip-Hop
15262122-71d7-4f9e-...
https://album...
Debussy: Cl...
1905
Claude Deb...
Piano Solo
Classical
1ef9b952-81d7-41c6-9...
https://album...
Future Nost...
2020
Dua Lipa
Warner Records
Pop
8f280921-db49-4ad3-...
https://album...
Beethoven: ...
1800
Ludwig van...
Full Orchestra (Stri...
Classical
97da17da-6fc7-413b-b...
https://album...
Vivaldi: The...
1723
Antonio Viv...
Chamber Orchestr...
Classical
d7aef3d1-5df9-4063-9...
https://album...
Astroworld
2019
Travis Scott...
Vocals, Synths, 80...
Hip-Hop
85428806-9d5a-4385-...
https://album...
Scorpion
2018
Drake
Vocals, Synths, 80...
Hip-Hop
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences



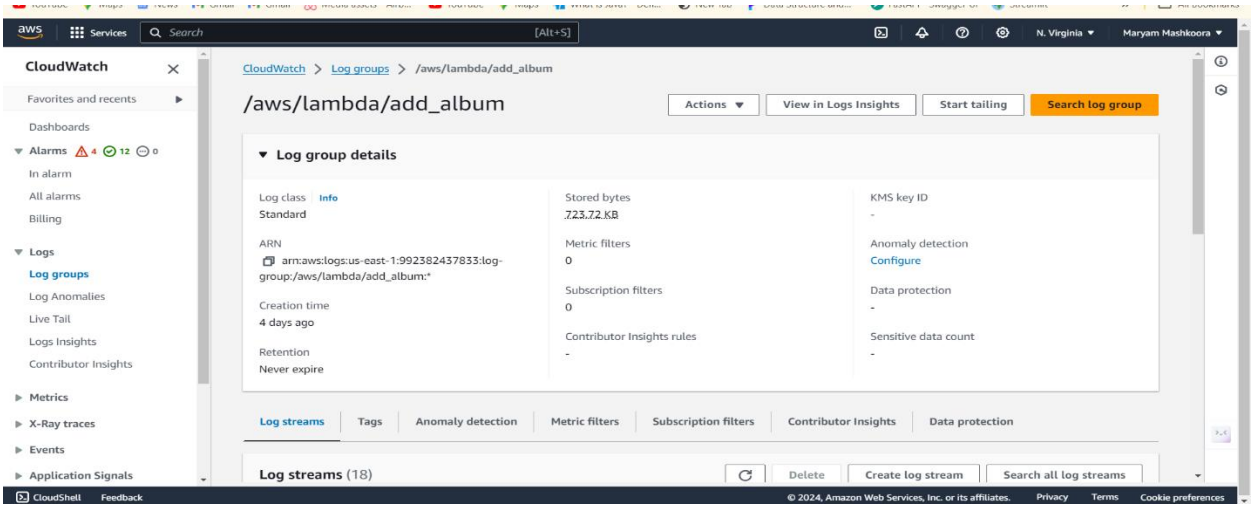

## 05) AWS IAM

Here, AWS IAM (Identity and Access Management) used by creating a role called “admin” to allow permission ensuring only admin can perform the sensitive CRUD operations and also this is given some policies like AmzonS3FullAccess, DynomoDbFullAccess, CloudWatchFullAccess where, this IAM role will be able to access a do the operation.



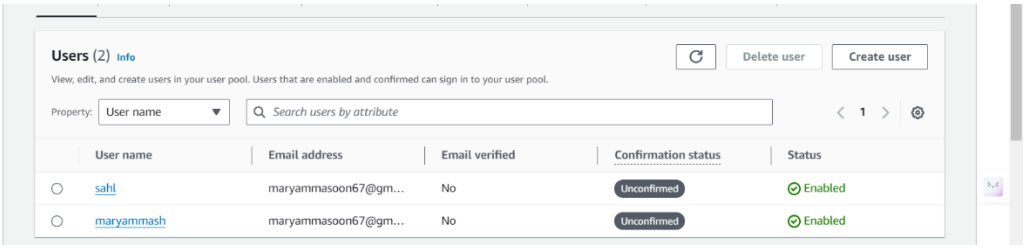
## 06) AWS CloudFront

The system uses the Amazon CloudFront to enhance the website's performance by delivering content with low latency and increases the speed and reliability of content delivery to employees while lowering latency and improving user experience.



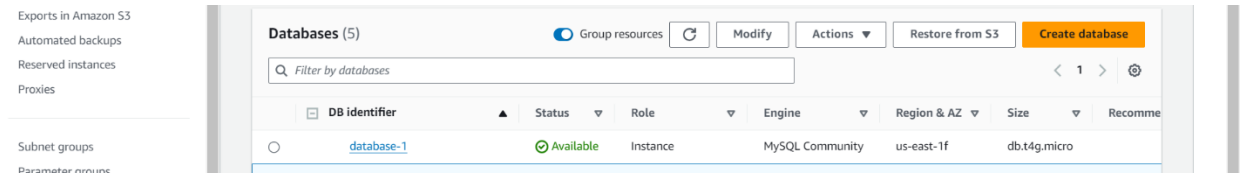

## 07) AWS Cognito

As Cognito Authenticate users and manage access tokens where its purpose is handles user sign-up, sign-in, ensuring that only authorized personnel can access or modify the applications and their data. This enhances the security and compliance of the system by providing fine-grained access control. AWS Cognito to implement secure user authentication for the DreamStreamer website. It allows admins to log in safely and manage permissions and User to log in safely to the User side.

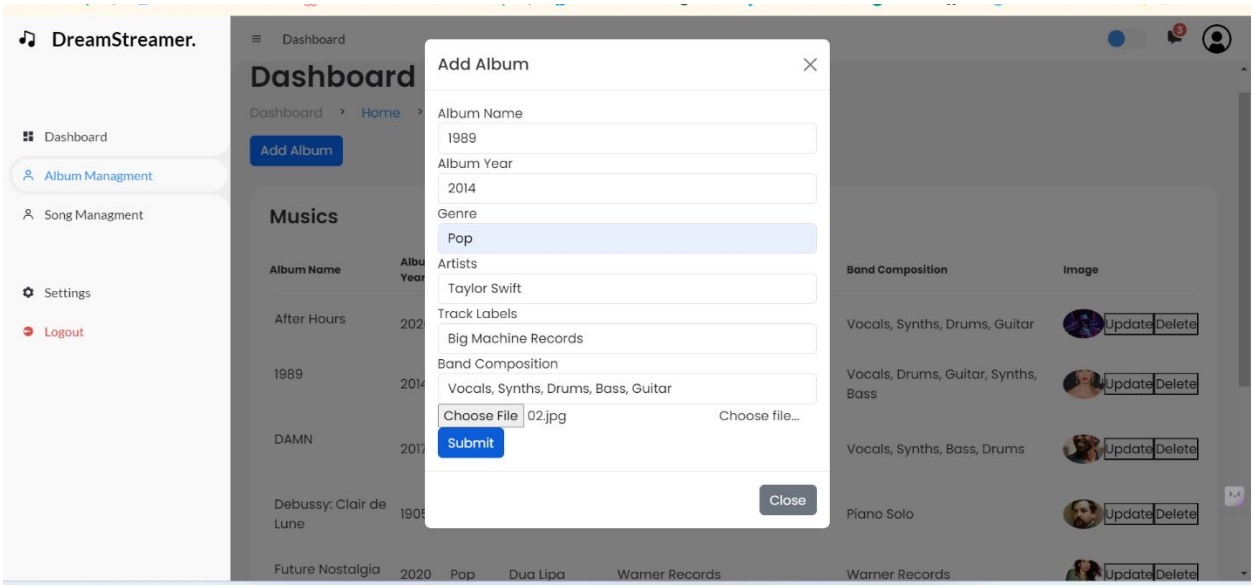



## 07) AWS RDS

This solution use RDS here to create a relational database to do the analytics where when ever if user purchase the id if the song and the purchased amount of that particular song increase by one and through the SNS the notification will be sent to admin to find the highest bout song.




# User Interface



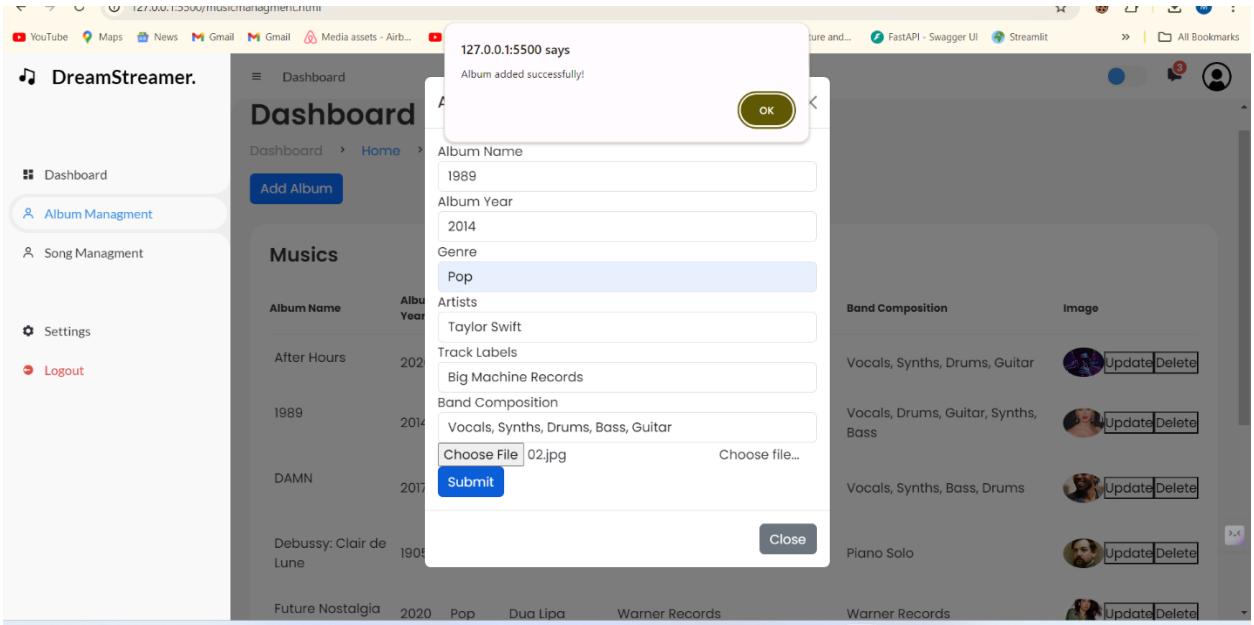




*Figure 7.Album added successfully*

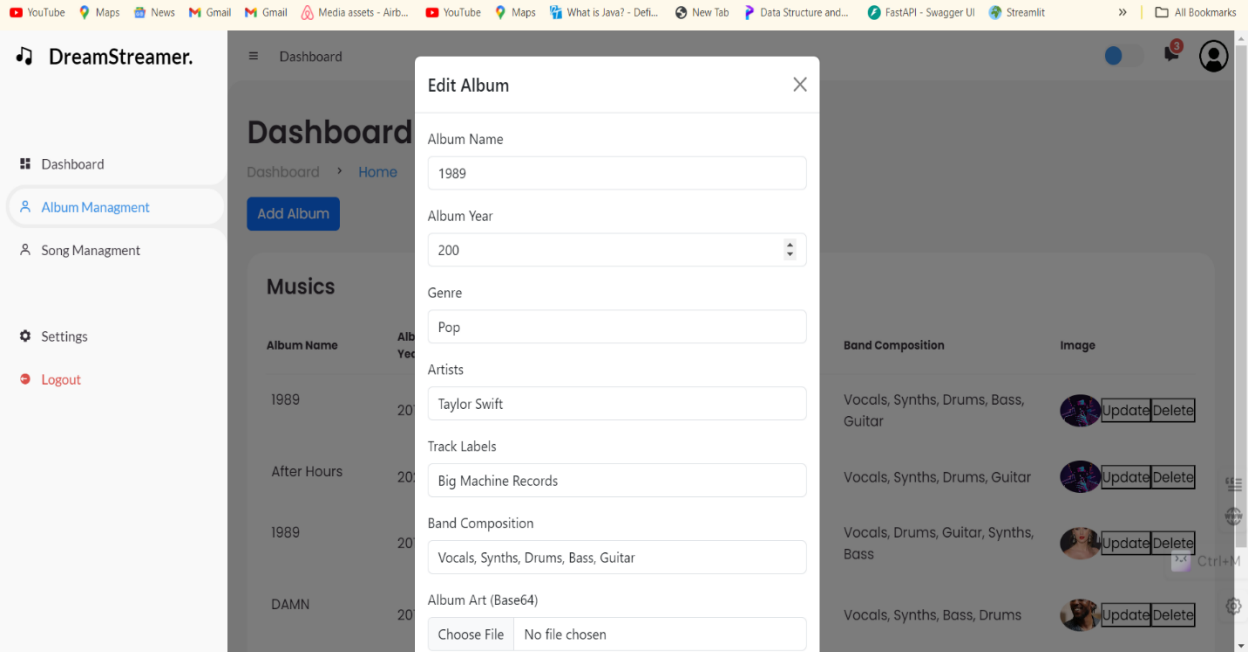



*Figure 8.Update Function for Album*

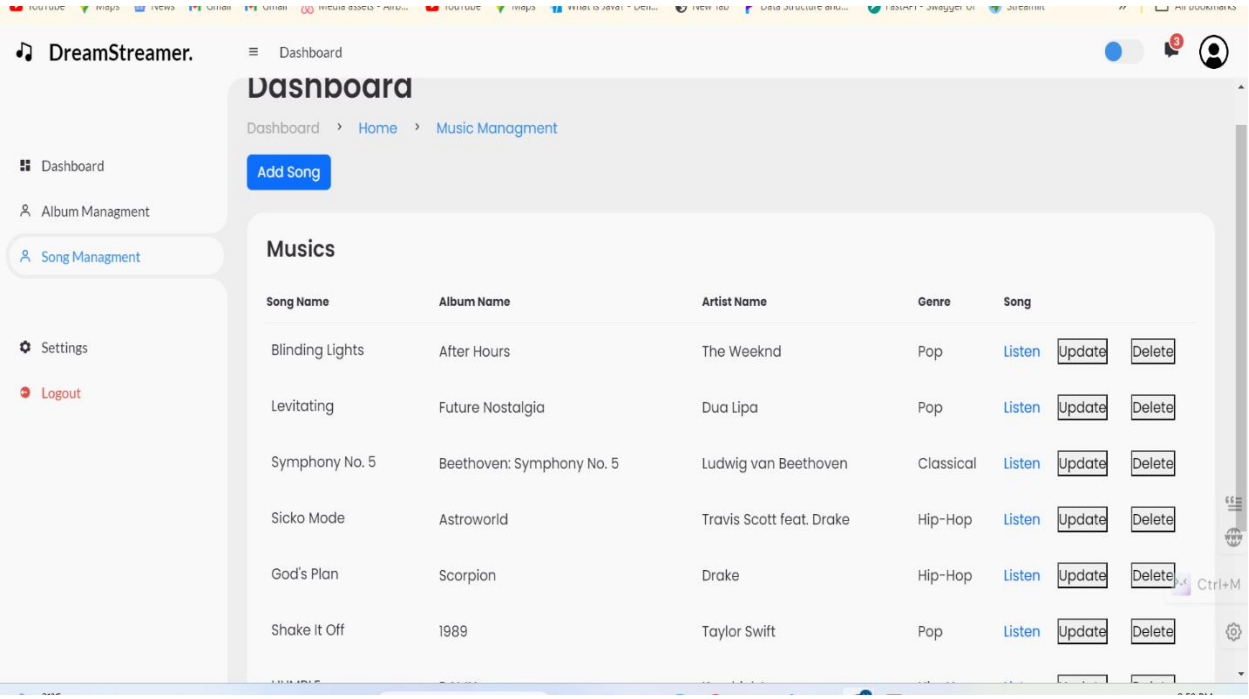



*Figure 9.Get song*

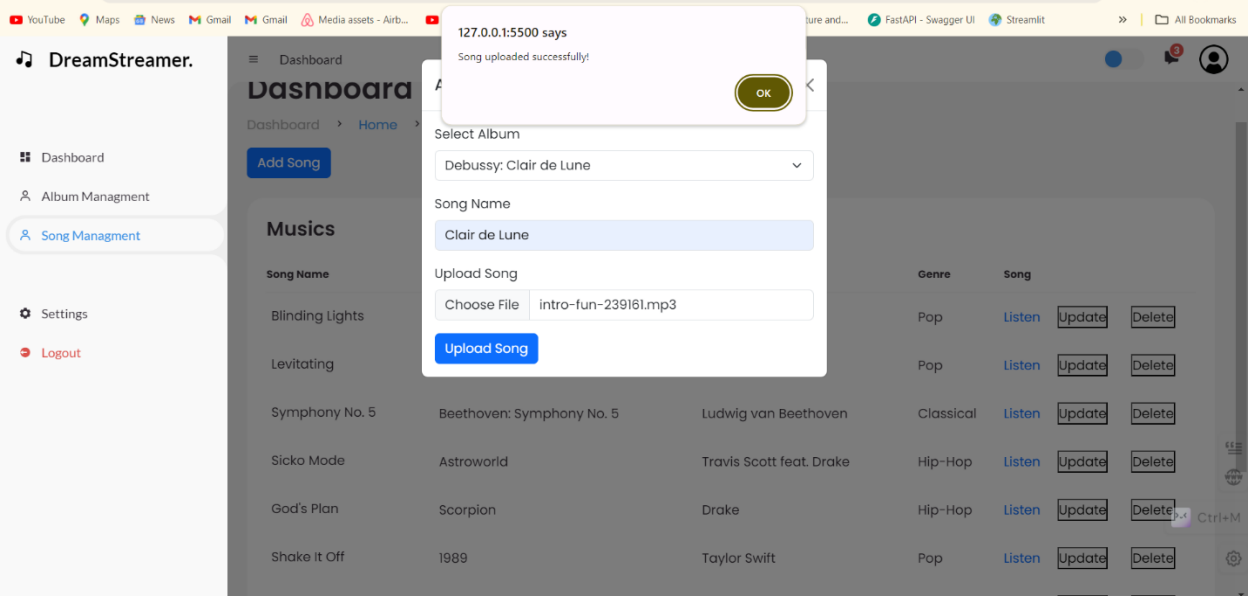



*Figure 10.Song Uploded*

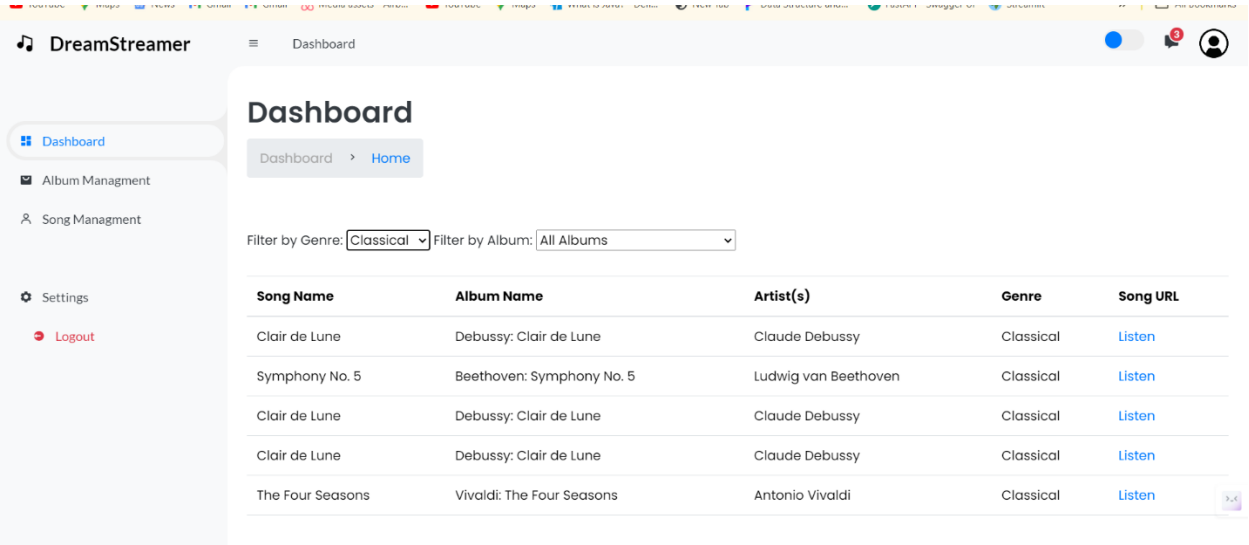



*Figure 11.Filter for admin using genre and albums*

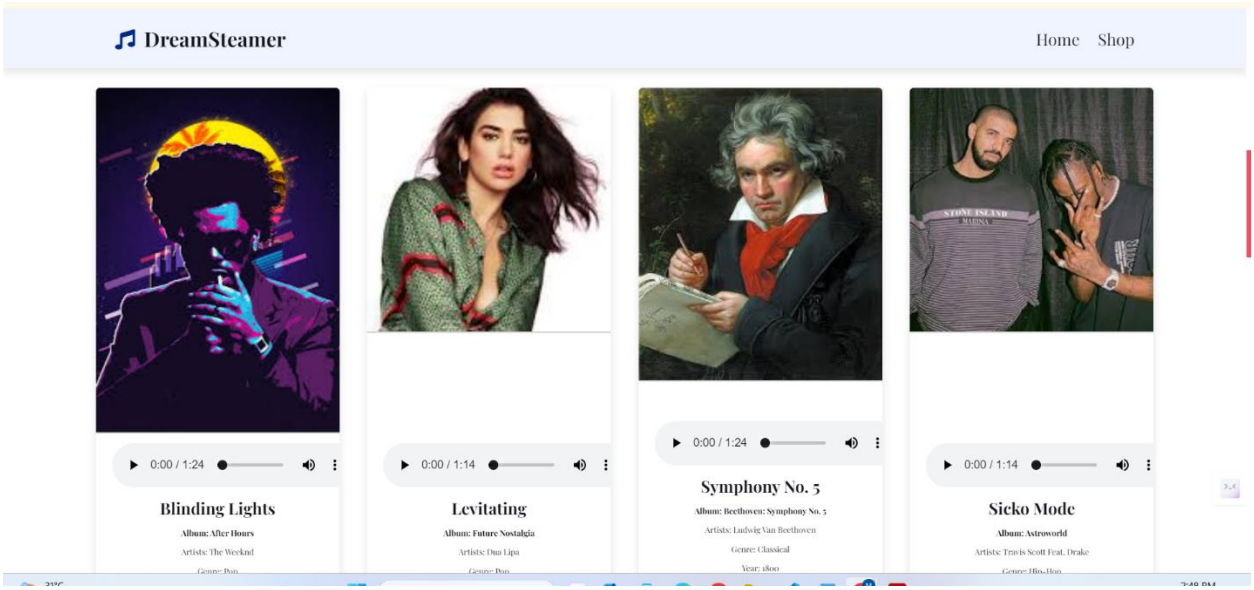



*Figure 12.User Interface to view Albums*

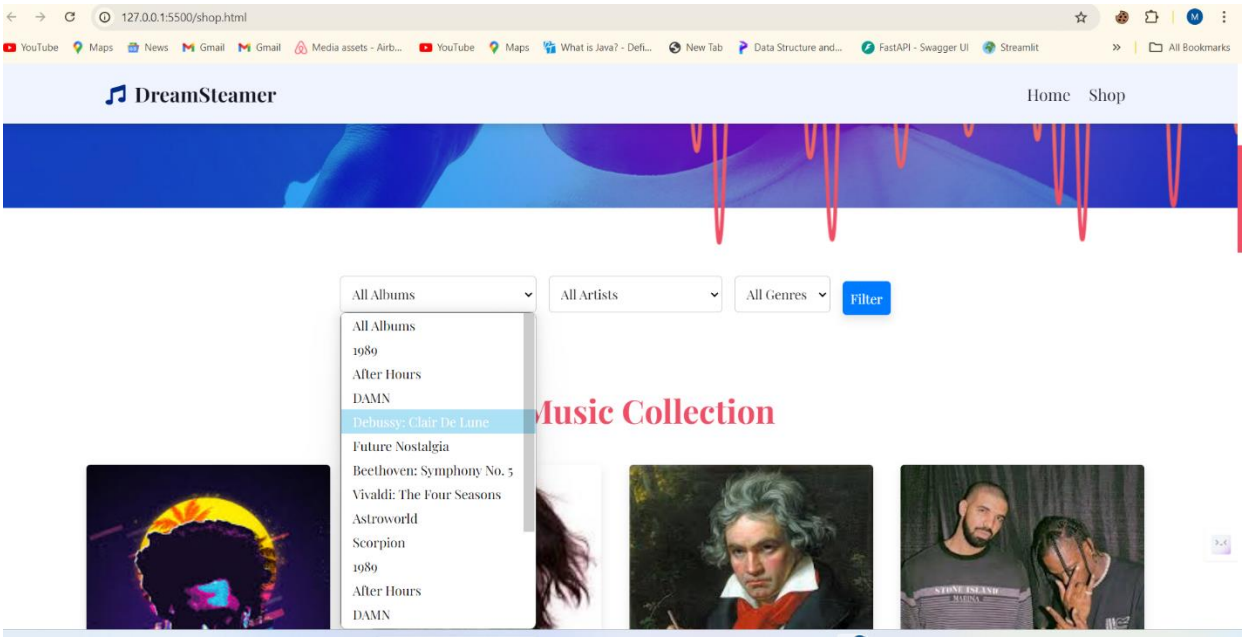



*Figure 13.User Filter System*

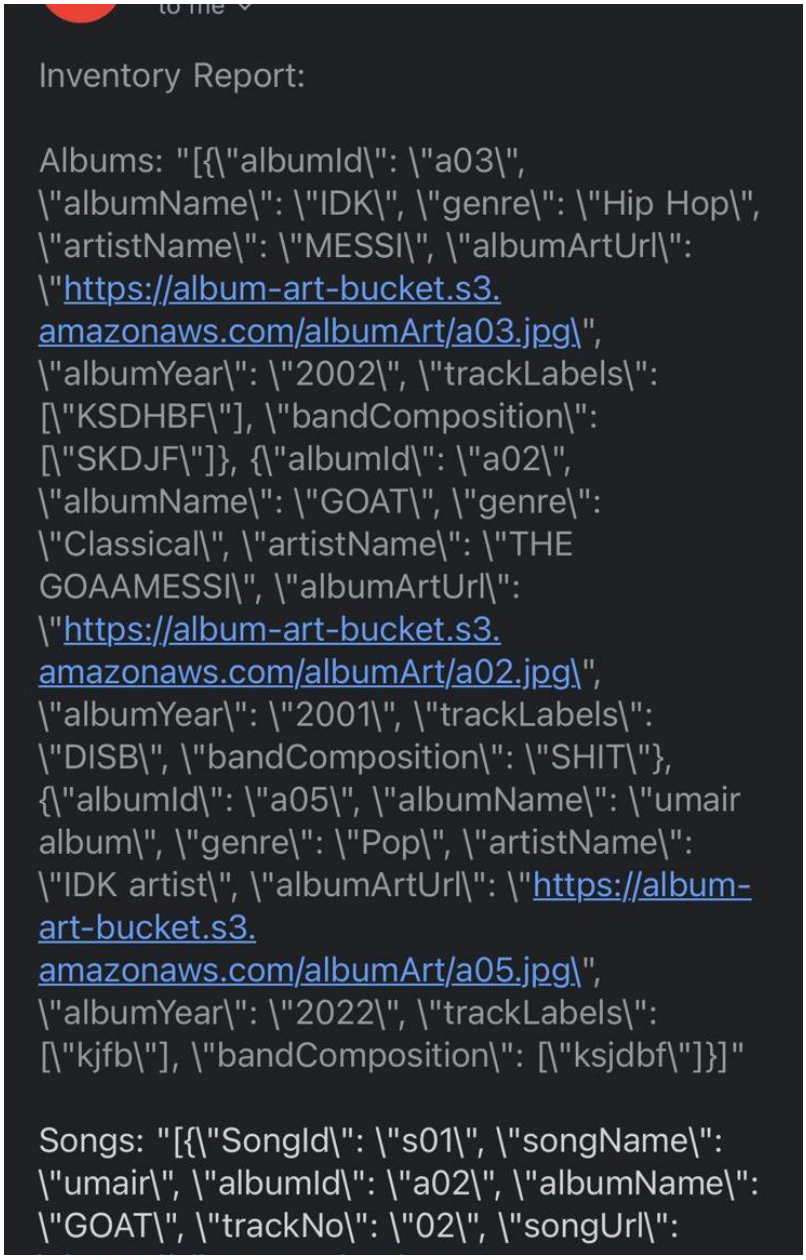



*Figure 14.SNS Email*

CB010658

# 3. Appendices

## Lamba Function of  Add Album

```
import json
import boto3
import uuid
from botocore.exceptions import ClientError
from decimal import Decimal
from base64 import b64decode

# Initialize DynamoDB and S3 resources
dynamodb = boto3.resource('dynamodb', region_name='us-east-1')
s3_client = boto3.client('s3')
dynamodb_table = dynamodb.Table('Albums')  # Your table name
S3_BUCKET = 'album-art1'  # Your S3 bucket name

def lambda_handler(event, context):
    print('Request event:', event)
    response = None

    try:
        # Extract the body from the event if wrapped in another object
        body = event.get('body', event)

        # Parse the body
        album_data = json.loads(body)

        # Handle image upload if provided
        if 'imageBase64' in album_data:
```

```
        image_url = upload_image_to_s3(album_data['imageBase64'], album_data.get('AlbumName', 'default_image'))
        album_data['AlbumArtURL'] = image_url

    response = save_album(album_data)

  except Exception as e:
    print('Error:', e)
    response = build_response(400, 'Error processing request')

  return response

def upload_image_to_s3(image_base64, album_name):
  try:
    # Decode the base64 image
    image_data = b64decode(image_base64)
    image_key = f"{album_name}_{uuid.uuid4()}.png"

    # Upload the image to S3
    s3_client.put_object(Bucket=S3_BUCKET, Key=image_key, Body=image_data, ContentType='image/jpeg')

    # Generate the S3 URL
    image_url = f"https://{S3_BUCKET}.s3.amazonaws.com/{image_key}"
    return image_url

  except ClientError as e:
    print('S3 Error:', e)
    raise Exception('Image upload failed')
```

CB010658

```
def save_album(album_data):
  try:
    # Use the provided id or generate a new one if not provided
    album_id = album_data.get('AlbumID', str(uuid.uuid4()))

    album_item = {
      'album_id': album_id,
      'AlbumName': album_data.get('AlbumName'),
      'AlbumYear': album_data.get('AlbumYear'),
      'Genre': album_data.get('Genre'),
      'Artists': album_data.get('Artists'),
      'TrackLabels': album_data.get('TrackLabels'),
      'BandComposition': album_data.get('BandComposition'),
      'AlbumArtURL': album_data.get('AlbumArtURL', '')  # Optional field
    }

    # Put the item in the DynamoDB table
    dynamodb_table.put_item(Item=album_item)

    body = {
      'Operation': 'SAVE',
      'Message': 'SUCCESS',
      'Item': album_item
    }
    return build_response(200, body)
  except ClientError as e:
    print('DynamoDB Error:', e)
    return build_response(400, e.response['Error']['Message'])
```

```
def build_response(status_code, body):
    return {
        'statusCode': status_code,
        'headers': {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*',
            'Access-Control-Allow-Headers': 'Content-Type',
            'Access-Control-Allow-Methods': 'OPTIONS,POST'
        },
        'body': json.dumps(body, cls=DecimalEncoder)
    }

class DecimalEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Decimal):
            if obj % 1 == 0:
                return int(obj)
            else:
                return float(obj)
        return super(DecimalEncoder, self).default(obj)
```

## Lambda function of  edit abum

```
import base64
import boto3
from botocore.exceptions import ClientError
from decimal import Decimal
```

CB010658

```
# Initialize AWS services clients
dynamodb = boto3.resource('dynamodb')
s3_client = boto3.client('s3')

# Name of the DynamoDB table and S3 bucket
table_name = 'Albums'
bucket_name = 'album-art1'

# Custom JSON encoder to handle Decimal types
class DecimalEncoder(json.JSONEncoder):
  def default(self, obj):
    if isinstance(obj, Decimal):
      return float(obj) if obj % 1 else int(obj)
    return super(DecimalEncoder, self).default(obj)

# Function to fix base64 padding
def fix_base64_padding(base64_string):
  # Add the correct padding if it's missing
  missing_padding = len(base64_string) % 4
  if missing_padding != 0:
    base64_string += '=' * (4 - missing_padding)
  return base64_string

def lambda_handler(event, context):
  try:
    # Parse the request body
    if isinstance(event.get('body'), str):
      event = json.loads(event['body'])
```

```
    # Extract data to update
album_id = event['album_id']
album_name = event.get('AlbumName')  # Using .get to handle optional fields
genre = event.get('Genre')
artist_name = event.get('Artists')
album_year = event.get('AlbumYear')
track_labels = event.get('TrackLabels')  # Assumed to be a list of strings
band_composition = event.get('BandComposition')
album_art_base64 = event.get('AlbumArtURL')  # Base64 encoded image data (optional)



# If new album art is provided, fix padding and upload to S3
album_art_url = None
if album_art_base64:
  # Fix base64 string padding if necessary
  album_art_base64 = fix_base64_padding(album_art_base64)

  # Decode the base64 string to bytes
  try:
    album_art_bytes = base64.b64decode(album_art_base64)
  except Exception as e:
    return {
      'statusCode': 400,
      'body': json.dumps({
        'message': 'Error decoding Base64 image data',
        'error': str(e)
      })
    }
```

```
        # Upload the decoded image bytes directly to S3
        s3_key = f"albumArt/{album_id}.jpg"  # Use AlbumID for the image filename
        try:
            s3_client.put_object(
                Bucket=bucket_name,
                Key=s3_key,
                Body=album_art_bytes,
                ContentType='image/png'
            )
            album_art_url = f"https://{bucket_name}.s3.amazonaws.com/{s3_key}"
        except ClientError as e:
            return {
                'statusCode': 500,
                'body': json.dumps({
                    'message': 'Error uploading image to S3',
                    'error': str(e)
                })
            }

    # Prepare the update expression for DynamoDB
    update_expression = "set "
    expression_attribute_values = {}

    # Only include fields that are provided in the request
    if album_name:
        update_expression += "AlbumName=:n, "
        expression_attribute_values[':n'] = album_name
    if genre:
        update_expression += "Genre=:g, "
```

```
        expression_attribute_values[':g'] = genre
    if artist_name:
        update_expression += "Artists=:a, "
        expression_attribute_values[':a'] = artist_name
    if album_year:
        update_expression += "AlbumYear=:y, "
        expression_attribute_values[':y'] = album_year
    if track_labels:
        update_expression += "TrackLabels=:t, "
        expression_attribute_values[':t'] = track_labels
    if band_composition:
        update_expression += "BandComposition=:b, "
        expression_attribute_values[':b'] = band_composition
    if album_art_url:
        update_expression += "AlbumArtURL=:u, "
        expression_attribute_values[':u'] = album_art_url

    # Remove trailing comma and space from update expression
    if update_expression.endswith(", "):
        update_expression = update_expression[:-2]

    # Ensure there are fields to update
    if not expression_attribute_values:
        return {
            'statusCode': 400,
            'body': json.dumps({
                'message': 'No attributes provided to update'
            })
        }
```

```
    # Get the table resource
    table = dynamodb.Table(table_name)

    # Update the album data in DynamoDB
    response = table.update_item(
      Key={'album_id': album_id},
      UpdateExpression=update_expression,
      ExpressionAttributeValues=expression_attribute_values,
      ReturnValues="UPDATED_NEW"
    )

    # Return success response, using DecimalEncoder to serialize Decimals
    return {
      'statusCode': 200,
      'headers': {
        'Access-Control-Allow-Origin': '*',
        'Access-Control-Allow-Methods': 'PUT, GET, POST, OPTIONS',
        'Access-Control-Allow-Headers': 'Content-Type'
      },
      'body': json.dumps({
        'message': 'Album updated successfully!',
        'updatedAttributes': response['Attributes']
      }, cls=DecimalEncoder)  # Using DecimalEncoder for Decimal conversion
    }

  except ClientError as e:
    return {
      'statusCode': 500,
```

```
            'body': json.dumps({
                'message': 'Error updating album',
                'error': str(e)
            })
        }

    except Exception as e:
        return {
            'statusCode': 500,
            'body': json.dumps({
                'message': 'Unexpected error occurred',
                'error': str(e)
            })
        }
```

## Lambda function Get Function

```
import json
import boto3
from decimal import Decimal
from botocore.exceptions import ClientError

# Initialize AWS services clients
dynamodb = boto3.resource('dynamodb')

# Name of the DynamoDB table
table_name = 'Albums'

# Custom JSON encoder to handle Decimal types
```

CB010658

```
class DecimalEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Decimal):
            # Convert Decimal to float or int
            return float(obj) if obj % 1 else int(obj)
        return super(DecimalEncoder, self).default(obj)

def lambda_handler(event, context):
    try:
        # Get the table resource
        table = dynamodb.Table(table_name)

        # Scan the table to get all album entries
        response = table.scan()
        albums = response.get('Items', [])

        # Return the list of albums with custom Decimal handling
        return {
            'statusCode': 200,
            'headers': {
                'Access-Control-Allow-Origin': '*',
                'Access-Control-Allow-Methods': 'POST, GET, OPTIONS',
                'Access-Control-Allow-Headers': 'Content-Type'
            },
            'body': json.dumps(albums, cls=DecimalEncoder)  # Use custom encoder here
        }

    except ClientError as e:
        print(f"Error: {e}")
```

```
    return {
      'statusCode': 500,
      'body': json.dumps({
        'message': 'Error fetching albums',
        'error': str(e)
      })
    }
```

## Lambda Function Delete Album

```
import json
import boto3
from botocore.exceptions import ClientError

# Initialize AWS services clients
dynamodb = boto3.resource('dynamodb')
s3_client = boto3.client('s3')

# Name of the DynamoDB table and S3 bucket
table_name = 'Albums'
bucket_name = 'album-art1'

def lambda_handler(event, context):
  try:
    # Retrieve the AlbumID from the path parameters
    album_id = event['pathParameters']['album_id']

    # Initialize the DynamoDB table
    table = dynamodb.Table(table_name)
```

```
    # Retrieve the album information from DynamoDB to get the S3 key for album art
    response = table.get_item(
        Key={'album_id': album_id}
    )

    # Check if the album exists
    if 'Item' not in response:
        return {
            'statusCode': 404,
            'headers': {
                'Access-Control-Allow-Origin': '*',
                'Access-Control-Allow-Methods': 'POST, GET, DELETE, OPTIONS',
                'Access-Control-Allow-Headers': 'Content-Type'
            },
            'body': json.dumps({
                'message': f'Album with ID {album_id} not found'
            })
        }

    # Get the S3 URL from the album item
    album_art_url = response['Item'].get('AlbumArtURL', None)

    # If album art exists, extract the S3 key and delete the album art from S3
    if album_art_url:
        # Extract the S3 key from the album art URL (after "https://bucket-
name.s3.amazonaws.com/")
        s3_key = album_art_url.split(f"https://{bucket_name}.s3.amazonaws.com/")[-1]

        # Delete the album art from the S3 bucket
```

```
        try:
            s3_client.delete_object(Bucket=bucket_name, Key=s3_key)
        except ClientError as e:
            return {
                'statusCode': 500,
                'headers': {
                    'Access-Control-Allow-Origin': '*',
                    'Access-Control-Allow-Methods': 'POST, GET, DELETE, OPTIONS',
                    'Access-Control-Allow-Headers': 'Content-Type'
                },
                'body': json.dumps({
                    'message': 'Error deleting album art from S3',
                    'error': str(e)
                })
            }

    # Delete the album record from DynamoDB
    table.delete_item(
        Key={'album_id': album_id}
    )

    # Return success response
    return {
        'statusCode': 200,
        'headers': {
            'Access-Control-Allow-Origin': '*',
            'Access-Control-Allow-Methods': 'POST, GET, DELETE, OPTIONS',
            'Access-Control-Allow-Headers': 'Content-Type'
        },
```

```
            'body': json.dumps({
                'message': f'Album with ID {album_id} deleted successfully'
            })
        }

    except ClientError as e:
        print(f"ClientError: {e}")
        return {
            'statusCode': 500,
            'headers': {
                'Access-Control-Allow-Origin': '*',
                'Access-Control-Allow-Methods': 'POST, GET, DELETE, OPTIONS',
                'Access-Control-Allow-Headers': 'Content-Type'
            },
            'body': json.dumps({
                'message': 'Error deleting album',
                'error': str(e)
            })
        }

    except Exception as e:
        print(f"Unexpected error: {e}")
        return {
            'statusCode': 500,
            'headers': {
                'Access-Control-Allow-Origin': '*',
                'Access-Control-Allow-Methods': 'POST, GET, DELETE, OPTIONS',
                'Access-Control-Allow-Headers': 'Content-Type'
            },
```

```
        'body': json.dumps({
            'message': 'Unexpected error occurred',
            'error': str(e)
        })
    }
```

## Lambda Function Add Songs

```
import json
import base64
import uuid  # To generate unique song_id
import boto3
from botocore.exceptions import ClientError

# Initialize AWS services clients
s3 = boto3.client('s3')
dynamodb = boto3.resource('dynamodb')

def lambda_handler(event, context):
    print("Received event:", json.dumps(event))  # Log the incoming event
    try:
        # Check if the event body is a string and needs decoding
        if 'body' in event:
            event_body = json.loads(event['body'])  # Parse the body if it's in a string format
        else:
            event_body = event  # Assume it's already parsed

        album_id = event_body['album_id']  # Use album_id directly from the event
        songs = event_body['songs']
```

```
    if not (1 <= len(songs) <= 5):
        return {
            'statusCode': 400,
            'body': json.dumps({
                'message': 'You must upload at least 1 song and at most 5 songs.'
            }),
            'headers': {
                'Access-Control-Allow-Origin': '*',
            }
        }

    # Step 1: Retrieve AlbumName and Artists from the Albums table using album_id
    albums_table = dynamodb.Table('Albums')
    album_response = albums_table.get_item(
        Key={'album_id': album_id}
    )

    if 'Item' not in album_response:
        return {
            'statusCode': 404,
            'body': json.dumps({
                'message': f'Album with id {album_id} not found.'
            }),
            'headers': {
                'Access-Control-Allow-Origin': '*',
            }
        }
```

```
    album_item = album_response['Item']
    album_name = album_item['AlbumName']
    artists = album_item['Artists']
    genre = album_item['Genre']

    # Prepare to upload songs and insert them into the Songs table
    s3_bucket = 'dreamstreamer-song1'  # Use your actual S3 bucket name
    song_urls = []
    songs_table = dynamodb.Table('Songs')

    # Step 2: Iterate through each song, upload it to S3, and add an entry in the Songs table
    for song in songs:
        song_name = song['song_name']
        song_base64 = song['song_data']
        song_base64 = fix_base64_padding(song_base64)
        song_bytes = base64.b64decode(song_base64)

        # Generate unique song_id and S3 key, including song name
        song_id = str(uuid.uuid4())
        sanitized_song_name = song_name.replace(" ", "_").lower()  # Sanitize song name for S3 key
        s3_key = f'songs/{album_id}/{sanitized_song_name}_{song_id}.mp3'  # Save songs in the 'songs/' folder

        # Upload song to S3
        s3.put_object(
            Bucket=s3_bucket,
            Key=s3_key,
            Body=song_bytes,
            ContentType='audio/mpeg'
```

```
        )

        # Generate the S3 URL for the uploaded song
        s3_url = f'https://{s3_bucket}.s3.amazonaws.com/{s3_key}'
        song_urls.append(s3_url)

        # Insert the song record into the Songs table
        songs_table.put_item(
            Item={
                'song_id': song_id,
                'album_id': album_id,
                'song_name': song_name,  # Store the song name in DynamoDB
                'song_url': s3_url,
                'AlbumName': album_name,  # Save the AlbumName
                'Artists': artists,       # Save the Artists
                'Genre': genre
            }
        )

    # Step 3: Return success response
    return {
        'statusCode': 200,
        'body': json.dumps({
            'message': 'Songs successfully uploaded!',
            'album_id': album_id,
            'song_urls': song_urls
        }),
        'headers': {
            'Access-Control-Allow-Origin': '*',
```

```
        }
    }

except ClientError as e:
    print(f"Error: {e}")
    return {
        'statusCode': 500,
        'body': json.dumps({
            'message': 'Error uploading songs',
            'error': str(e)
        }),
        'headers': {
            'Access-Control-Allow-Origin': '*',
        }
    }

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

CB010658

```
# Function to fix base64 padding
def fix_base64_padding(base64_string):
    missing_padding = len(base64_string) % 4
    if missing_padding != 0:
        base64_string += '=' * (4 - missing_padding)
    return base64_string
```

## Lambda Function Edit Songs

```
import json
import base64
import boto3
from botocore.exceptions import ClientError

# Initialize AWS services clients
s3 = boto3.client('s3')
dynamodb = boto3.resource('dynamodb')

def lambda_handler(event, context):
    print("Received event:", json.dumps(event))  # Log the incoming event
    try:
        # Check if the event body is a string and needs decoding
        if 'body' in event:
            event_body = json.loads(event['body'])  # Parse the body if it's in a string format
        else:
            event_body = event  # Assume it's already parsed

        song_id = event_body['song_id']  # The ID of the song to edit
        album_id = event_body['album_id']  # The album ID the song belongs to
        new_song_name = event_body.get('song_name')  # New song name (if changing)
```

```
    new_song_data = event_body.get('song_data')  # New song data (if changing)

    songs_table = dynamodb.Table('Songs')

    # Step 1: Retrieve the existing song details
    song_response = songs_table.get_item(Key={'song_id': song_id})

    if 'Item' not in song_response:
      return {
        'statusCode': 404,
        'body': json.dumps({
          'message': f'Song with id {song_id} not found.'
        }),
        'headers': {
          'Access-Control-Allow-Origin': '*',
        }
      }

    song_item = song_response['Item']
    current_song_name = song_item['song_name']
    s3_bucket = 'dreamstreamer-song1'  # Your actual S3 bucket name
    s3_key = f'songs/{album_id}/{current_song_name.replace(" ", "_").lower()}_{song_id}.mp3'

    # Step 2: Update song details in DynamoDB
    update_expression = "SET"
    expression_attribute_values = {}

    if new_song_name:
      update_expression += " song_name = :new_name,"
```

```
        expression_attribute_values[":new_name"] = new_song_name

    # If new song data is provided, upload it to S3
    if new_song_data:
        # Decode and upload the new song file
        new_song_base64 = fix_base64_padding(new_song_data)
        new_song_bytes = base64.b64decode(new_song_base64)

        # Upload new song to S3
        s3.put_object(
            Bucket=s3_bucket,
            Key=s3_key,
            Body=new_song_bytes,
            ContentType='audio/mpeg'
        )
        # Update the song URL
        s3_url = f'https://{s3_bucket}.s3.amazonaws.com/{s3_key}'
        update_expression += " song_url = :new_url,"
        expression_attribute_values[":new_url"] = s3_url

    # Remove trailing comma
    update_expression = update_expression.rstrip(',')

    # Execute the update
    songs_table.update_item(
        Key={'song_id': song_id},
        UpdateExpression=update_expression,
        ExpressionAttributeValues=expression_attribute_values
    )
```

```
        # Step 3: Return success response
        return {
            'statusCode': 200,
            'body': json.dumps({
                'message': 'Song successfully updated!',
                'song_id': song_id,
                'new_song_name': new_song_name if new_song_name else current_song_name,
                'song_url': s3_url if new_song_data else song_item['song_url']
            }),
            'headers': {
                'Access-Control-Allow-Origin': '*',
            }
        }

    except ClientError as e:
        print(f"Error: {e}")
        return {
            'statusCode': 500,
            'body': json.dumps({
                'message': 'Error updating song',
                'error': str(e)
            }),
            'headers': {
                'Access-Control-Allow-Origin': '*',
            }
        }

    except Exception as e:
```

```
        print(f"Unexpected error: {e}")
        return {
            'statusCode': 500,
            'body': json.dumps({
                'message': 'Unexpected error occurred',
                'error': str(e)
            }),
            'headers': {
                'Access-Control-Allow-Origin': '*',
            }
        }

# Function to fix base64 padding
def fix_base64_padding(base64_string):
    missing_padding = len(base64_string) % 4
    if missing_padding != 0:
        base64_string += '=' * (4 - missing_padding)
    return base64_string
```

## Lambda Function to GET songs

```
import json
import boto3
from decimal import Decimal
from botocore.exceptions import ClientError

# Initialize AWS services clients
dynamodb = boto3.resource('dynamodb')

# Name of the DynamoDB table
```

CB010658

```
table_name = 'Songs'


# Custom JSON encoder to handle Decimal types
class DecimalEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Decimal):
            # Convert Decimal to float or int
            return float(obj) if obj % 1 else int(obj)
        return super(DecimalEncoder, self).default(obj)


def lambda_handler(event, context):
    try:
        # Get the table resource
        table = dynamodb.Table(table_name)

        # Scan the table to get all album entries
        response = table.scan()
        songs = response.get('Items', [])

        # Return the list of albums with custom Decimal handling
        return {
            'statusCode': 200,
            'headers': {
                'Access-Control-Allow-Origin': '*',
                'Access-Control-Allow-Methods': 'POST, GET, OPTIONS',
                'Access-Control-Allow-Headers': 'Content-Type'
            },
            'body': json.dumps(songs, cls=DecimalEncoder)  # Use custom encoder here
        }
```

```
    except ClientError as e:
        print(f"Error: {e}")
        return {
            'statusCode': 500,
            'body': json.dumps({
                'message': 'Error fetching albums',
                'error': str(e)
            })
        }
```

## Lambda Function

```
import json
import pymysql
import os

# Set your RDS credentials
rds_host = 'dreamstreamer1.c3aceemua9yn.eu-north-1.rds.amazonaws.com'
rds_username = 'admin'
rds_password = 'dreamstreamer1'
rds_db_name = 'dreamstreamer1'

def lambda_handler(event, context):
    # Check if event has a 'body' (when using API Gateway)
    if 'body' in event:
        # Parse the event body, assuming it's a JSON string
        body = json.loads(event['body'])
        album_id = body.get('album_id')
```

```
        user_id = body.get('user_id')
    else:
        # If not using API Gateway, directly extract album_id and user_id
        album_id = event.get('album_id')
        user_id = event.get('user_id')

    # Validate inputs
    if not album_id or not user_id:
        return {
            'statusCode': 400,
            'headers': {
                'Access-Control-Allow-Origin': '*',  # Allow CORS
                'Access-Control-Allow-Headers': 'Content-Type',
            },
            'body': json.dumps('album_id and user_id are required')
        }

    # Connect to the RDS MySQL database
    connection = None
    try:
        connection = pymysql.connect(
            host=rds_host,
            user=rds_username,
            password=rds_password,
            database=rds_db_name,
            cursorclass=pymysql.cursors.DictCursor
        )

        with connection.cursor() as cursor:
```

```
        # SQL query to insert data
        insert_query = """
        INSERT INTO purchases (album_id, user_id)
        VALUES (%s, %s);
        """

        # Execute the query
        cursor.execute(insert_query, (album_id, user_id))

        # Commit the transaction
        connection.commit()

    return {
        'statusCode': 200,
        'headers': {
            'Access-Control-Allow-Origin': '*',  # Allow CORS
            'Access-Control-Allow-Headers': 'Content-Type',
        },
        'body': json.dumps(f"Purchase saved successfully for album_id: {album_id} and user_id: {user_id}")
    }

  except Exception as e:
    return {
        'statusCode': 500,
        'headers': {
            'Access-Control-Allow-Origin': '*',  # Allow CORS
            'Access-Control-Allow-Headers': 'Content-Type',
        },
```

```
        'body': json.dumps(f"Error saving purchase: {str(e)}")
    }

  finally:
    if connection:
      connection.close()
```

## Lambda Function For SNS

```
import json
import boto3
from botocore.exceptions import ClientError

# Initialize SNS client
sns_client = boto3.client('sns')

def lambda_handler(event, context):
  try:
    # Extract inventory details from the incoming event
    inventory_details = event.get('inventoryDetails', {})

    # Build the message to send
    message = f"Inventory Report:\n\nAlbums: {json.dumps(inventory_details.get('albums', []), indent=2)}\n\nSongs: {json.dumps(inventory_details.get('songs', []), indent=2)}"

    # SNS publish parameters
    params = {
      'Message': message,
      'Subject': 'Inventory Report',
```

```
        'TopicArn': 'arn:aws:sns:us-east-1:891377037536:dreamstreamer_maryam'
    }

    # Send the SNS message
    response = sns_client.publish(**params)

    # Return success response
    return {
        'statusCode': 200,
        'body': json.dumps({
            'message': 'SNS message sent successfully!',
            'snsResponse': response
        })
    }

except ClientError as e:
    # Log and return the error
    return {
        'statusCode': 500,
        'body': json.dumps({
            'message': 'Error sending SNS message',
            'error': str(e)
        })
    }
```

# Conclusion

The incorporation of AWS technologies represents a thorough change of the university's IT infrastructure. The institution expects to gain significant scalability, security, and cost-effectiveness benefits by utilizing services such as EC2, RDS, S3, Elastic Beanstalk, IAM, and CloudWatch. Each diagram addresses a distinct need, delivering optimal performance and dependability for important applications and services.

Furthermore, the use of AWS's tiered storage and pay-as-you-go pricing structures promises significant cost savings by moving away from traditional capital investment approaches. This flexibility in resource distribution based on actual usage has the potential to provide large savings.

This deliberate shift to AWS not only improves current service capabilities for students and staff, but it also streamlines operations and reduces IT overheads. Such a forward-thinking approach prepares the university for future technology developments and increased demand. It is a complete overhaul that not only addresses urgent needs but also lays the groundwork for future innovation and growth.

References

‘Introduction to cloud computing and AWS’ (2019) *AWS Certified Solutions Architect Study Guide*, pp. 1–19. doi:10.1002/9781119560395.ch1

Kyriakou, N. *et al.* (2023) ‘Achieving seamless migration to private-cloud infrastructure for multi-campus universities’, *International Journal on Cloud Computing: Services and Architecture*, 13(2), pp. 25–38. doi:10.5121/ijccsa.2023.13202.

Modi, C. *et al.* (2012) ‘A survey on security issues and solutions at different layers of cloud computing’, *The Journal of Supercomputing*, 63(2), pp. 561–592. doi:10.1007/s11227-012-0831-5.