A·P·I·I·T

Inspire love for learning

STAFFORDSHIRE UNIVERSITY

Level 6

COMP60010: Enterprise Cloud and Distributed Applications - 2

Student Name: Pradikshan Balasubramaniam

Student Number: CB010280_21035799/1

Batch Code: HFK2422COM

Module Instructor: Mr. Nisala Aloka Bandara

Submission Date: 23rd September 2024

Contents

List of Figures

List of Tables

Abstract

This report presents the proof-of-concept solution designed for Dreamstreamer, an AI based music streaming platform. The primary objective of this assignment is to development a **'customer facing'** application which displays the services offered by Dreamstreamer ,and an **'admin portal'** for administrative tasks such as performing **CRUD (Create, Read, Update, Delete) operations** on albums and songs provided by Dreamstreamer, and also for analytic purposes such as keeping track of popular songs available in the Dreamstreamer catalogue.

Therefore, the report details the technical architecture and the implementation approach for the required proof-of-concept for Dreamstreamer.

Chapter 1

Introduction

1.1 Overview

DreamStreamer is an AI-driven music streaming service that provides personalized music recommendations based on user preferences. The goal of this project is to develop a proof-of-concept website hosted on Amazon Web Services (AWS) to showcase DreamStreamer's offerings. The website will visually represent the available music and artist information, and allow users to filter albums by various criteria such as genre, artist, and tracks.

In addition, an admin portal will be developed to enable DreamStreamer's administrators to perform CRUD operations on the albums and songs database, with secure authentication and the ability to generate inventory reports.

1.1.1 Key Features and Functionalities of the Customer Facing Web Application

The customer-facing web application is designed with two distinct user interfaces for users who do not have an account and those who are logged in.

The key features and functionalities of the web application for users without an user account

are as follows:

- **Home Page:** Includes a hero section showcasing DreamStreamer's features, offerings and testimonials.
- **Features Page:** A detailed description of DreamStreamer's features.
- **Pricing Page:** Displays various subscription plans and pricing models.
- **About and Support Page:** Provides information about DreamStreamer's timeline, FAQ section and support channels for assistance.
- **Sign-Up Page:** Allows users to create a new account by providing necessary information.
- **Login Page:** Enables existing users to sign in and access personalized features.

The key features and functionalities of the web application for users with an user account are as follows:

- **Songs Page:** Displays all the songs available on DreamStreamer. Each song can be played, and users can view detailed information such as song name, genre, album, and artist.
- **Albums Page:** Showcases all the albums available on DreamStreamer allowing users to browse through various albums and the respective songs available for each album.
- **Artists Page:** Lists all the artists affiliated with DreamStreamer.
- **Profile Page:** Allows users to view their profile information. Some attributes, such as display name or preferences, can be edited directly from this page.
- **Support Page:** Provides information about support channels and a FAQ section.

1.2 General Assumptions

The following assumptions were made when developing the solution:

- Customers are able to sign up and create their own accounts to access the application. However, admin accounts are pre-created in the admin user pool, and no new admin accounts can be registered through the application. Admins can only log in using existing credentials.
- Each song in the database includes its own artist name and genre fields, as songs within the same album may feature different artists or belong to varying genres.
- Popular songs are determined by which songs are clicked on the most.
- Popular albums are determined by which albums are purchased the most.

Chapter 2

Solution Overview and Design

2.1 Solution Design

The implementation of the Dreamstreamer customer-facing website and admin portal requires the following resources:

- Compute
- Storage
- Database
- Networking & Content Delivery
- Analytics

The following table (table 2.1) provides an overview of all the AWS services used for the development of the Dreamstreamer implementation.

Resource Type	**AWS Service**	**Description**
Compute	AWS Lambda	Serverless compute service that runs code in response to events
Database	AWS DynamoDB	Provides a NoSQL, serverless database service
	AWS Relational Database Service (RDS)	Managed relational database supporting various database engines
Storage	AWS Simple Storage Service (S3)	Provides scalable object storage for any type of data
Networking & Content Delivery	AWS API Gateway	Managed service for creating, publishing, and securing APIs
	AWS Cloudfront	Content Delivery Network (CDN) for fast delivery of data globally
	AWS VPC	Securely connects on-premises networks to AWS VPCs
Application Integration	AWS Simple Notification Service (SNS)	SNS is a fully managed messaging service that allows you to send notifications and messages to distributed systems and users

Security, Identity & Compliance	AWS Identity and Access Management (IAM)	Managed access to AWS services and resources securely
	AWS Web Application Firewall	AWS WAF protects web applications from common web exploits
	AWS Cognito	User authentication, authorization, and user management service

Table 2.1: Overview of all the AWS services needed for the development of the Dreamstreamer implementation

2.1.1 Explanation of the Cloud Services Utilized in the Solution

AWS Service	**Description of usage in the solution**
AWS Lambda	AWS Lambda is used to run serverless functions for handling CRUD operations. It manages data retrieval and updates between the database and the frontend for both the customer website and the admin portal. It also used to manage authentication tasks such as signup email verification.
AWS DynamoDB	AWS DynamoDB is used to store details of albums and songs.
AWS Relational Database Service (RDS)	AWS RDS is used to store analytics data, such as tracking which albums or songs are popular.

AWS Simple Storage Service (S3)	AWS S3 is used to host the static content of both the customer-facing website and the admin portal. It is also used to store album cover images and song audio files.
AWS API Gateway	AWS API Gateway is used to expose APIs for the frontend to communicate with the backend. It routes requests to AWS Lambda functions, enabling CRUD operations for both the customer site and the admin portal.
AWS Cloudfront	AWS CloudFront is used to improve website performance by serving the static content hosted in S3 through a content delivery network (CDN), optimizing load times for global users.
AWS VPC	AWS VPC is used to securely host AWS RDS within a private network. It also manages network configurations for a Lambda function that interacts with RDS, ensuring security and controlled access.
AWS Simple Notification Service (SNS)	AWS SNS is used to send notifications, such as the inventory report to the admin's email. It automates the report generation and delivery process based on admin requests from the portal.
AWS Identity and Access Management (IAM)	AWS IAM is used to manage access and permission policies for lambda functions and other services. It ensures that only authorized entities can perform operations like accessing databases or invoking APIs.

AWS Web Application Firewall (WAF)	AWS WAF is integrated with Amazon CloudFront to provide enhanced security for the Dreamstreamer web applications. It protects against common web threats, such as SQL injection and cross-site scripting. This integration ensures that malicious traffic is blocked before it reaches the application, safeguarding resources and improving performance.
AWS Cognito	AWS Cognito is used to manage user authentication for both customer accounts and admin login. It integrates a secure identity system, allowing users to create accounts and log in, while admins can authenticate before accessing sensitive data and reports.

Table 2.2: Explanation of the usage of all AWS services implemented in the solution

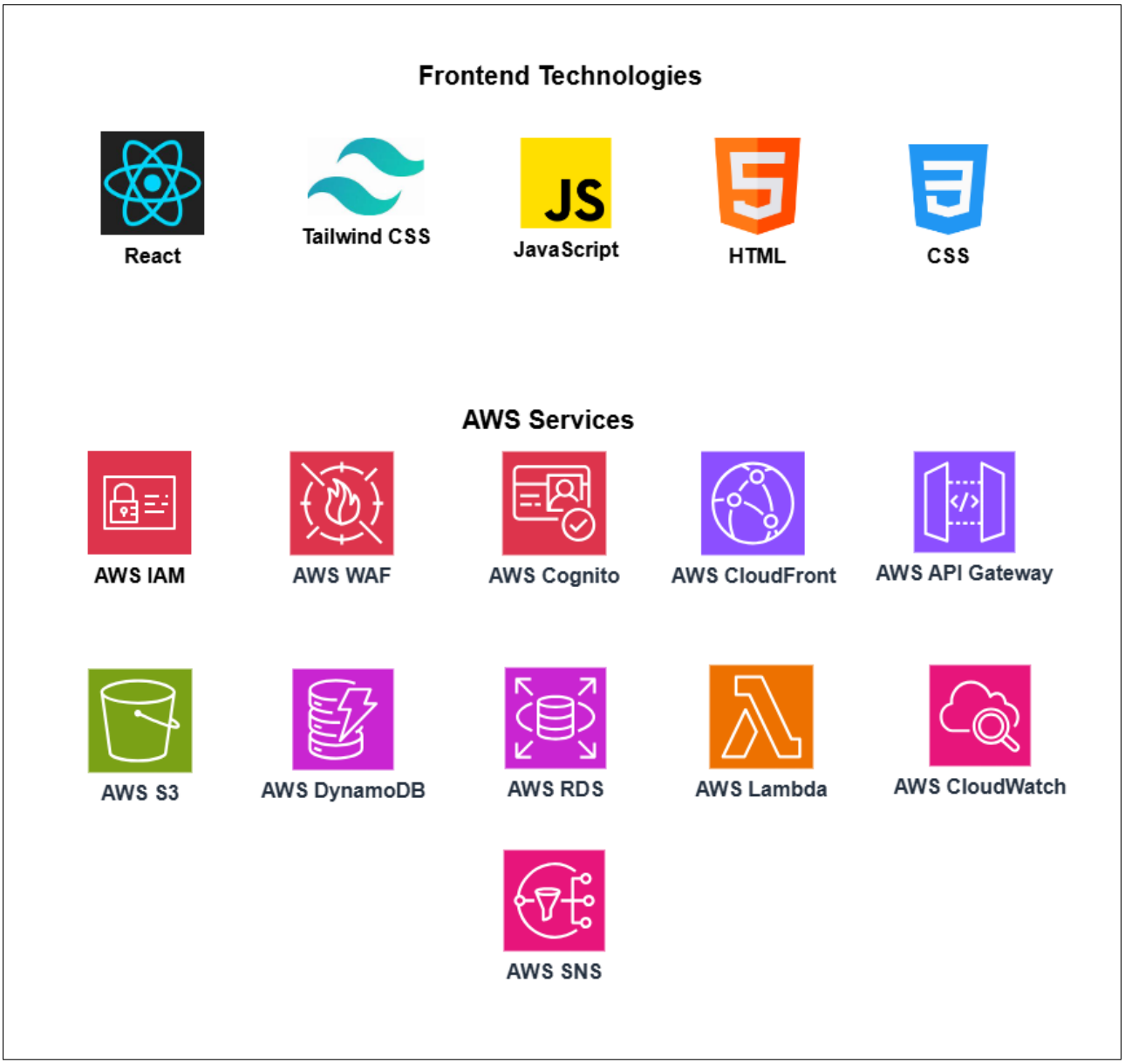


Figure 2.1: Frontend techlogies and AWS services used in the implementation

2.2 Solution Design Diagrams

2.2.1 Solution Design Diagram - Customer Facing Website

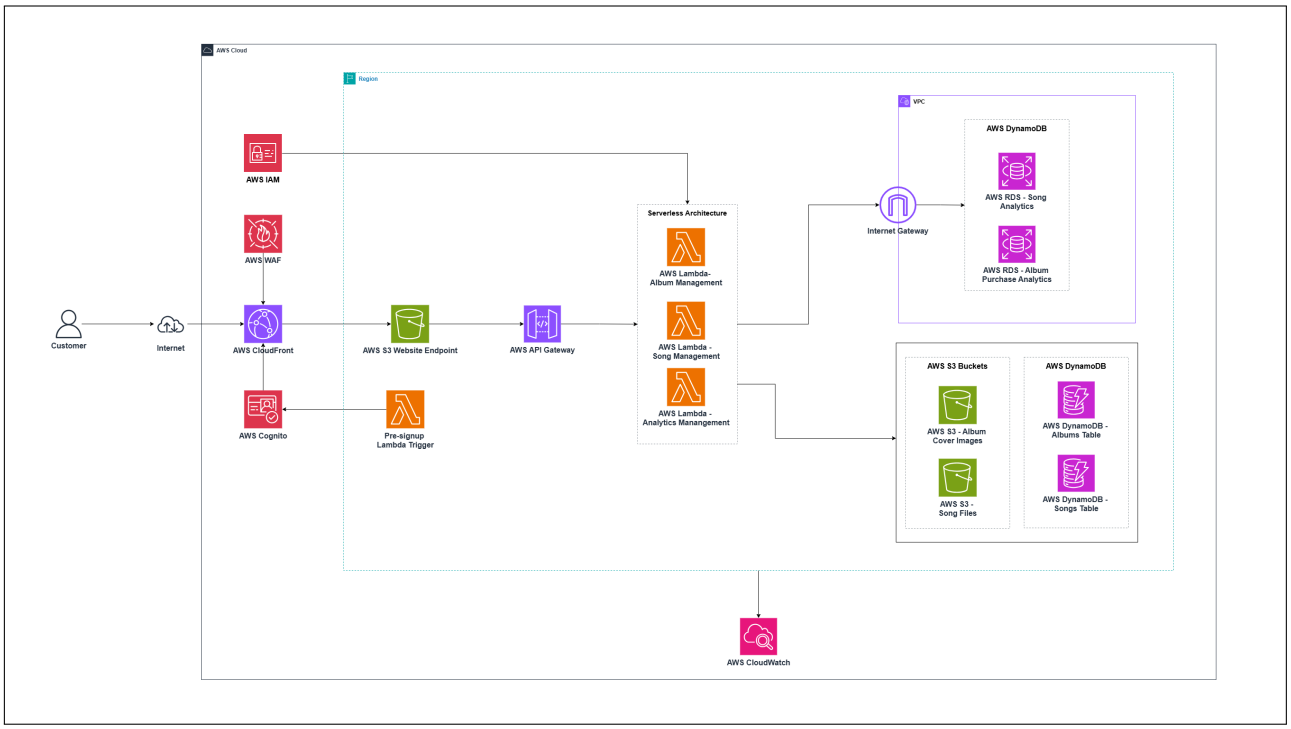


Figure 2.2: AWS Solution Design Diagram for Customer Facing Website (Refer to figure A.1 in appendix for enlarged diagram)

Workflow and Solution Design Explanation - Customer Facing Website

Component	Service	Function	Connections	Purpose
Frontend Access	CloudFront	Distributes content globally with low latency	Connected to customer cognito user pool, AWS WAF, and S3 website endpoint	Acts as a CDN, optimizing the delivery of static website, handles security via WAF, and user auth via Cognito

User Authentication	Cognito (Customer Pool)	Handles customer login and signup	Connected to CloudFront for customer access and with a pre-signup lambda trigger for user email validation	Manages user authentication, allowing only authorized users to access secure parts of the website
Security	AWS WAF	Protects from web attacks and threats	Integrated with CloudFront	Filters and blocks harmful requests to the customer website
Serverless Architecture	Lambda Functions	Manages backend logic by connecting the frontend to the backend	Connected to API Gateway, IAM, S3 Buckets, and RDS	Provides the backend logic without managing servers, handling requests from API Gateway, interacting with S3 and RDS
File Storage (Static)	S3 (Customer Website)	Hosts the static website	Connected to CloudFront and API Gateway	Stores static assets like HTML, CSS, and JavaScript files for the website
File Storage (Song and Album Data)	S3 (Song and Album Storage)	Stores album cover images and song audio files	Connected to lambda Functions and API Gateway	Allows lambda functions to store and retrieve album-related media

API Gateway	API Gateway	Routes requests from frontend to backend	Connected to CloudFront, lambda Functions, and S3 Buckets	Facilitates communication between the frontend and backend
IAM Roles	IAM	Manages permissions for Lambda functions	Connected to Lambda Functions, S3 Buckets, and RDS	Ensures Lambda functions have the necessary permissions to access other AWS services like S3 and RDS
Presignup Validation	Lambda function (Presignup)	Validates user email before signing up by checking if email exists in the userpool	Connected to Cognito customer user pool	Ensures validation logic for new users during signup
Data Storage (Analytics)	RDS	Stores analytics data related to album interactions	Connected to Lambda Functions and IAM	Stores analytical data about users and their interactions with albums, accessed by Lambda functions for data processing
Monitoring	CloudWatch	Monitors AWS service health and logs	Connected to all services in the AWS Region	Tracks and logs events, errors, and performance of the services

Table 2.3: Explanation of the workflow and solution design

2.2.2 Solution Design Diagram - Admin Portal

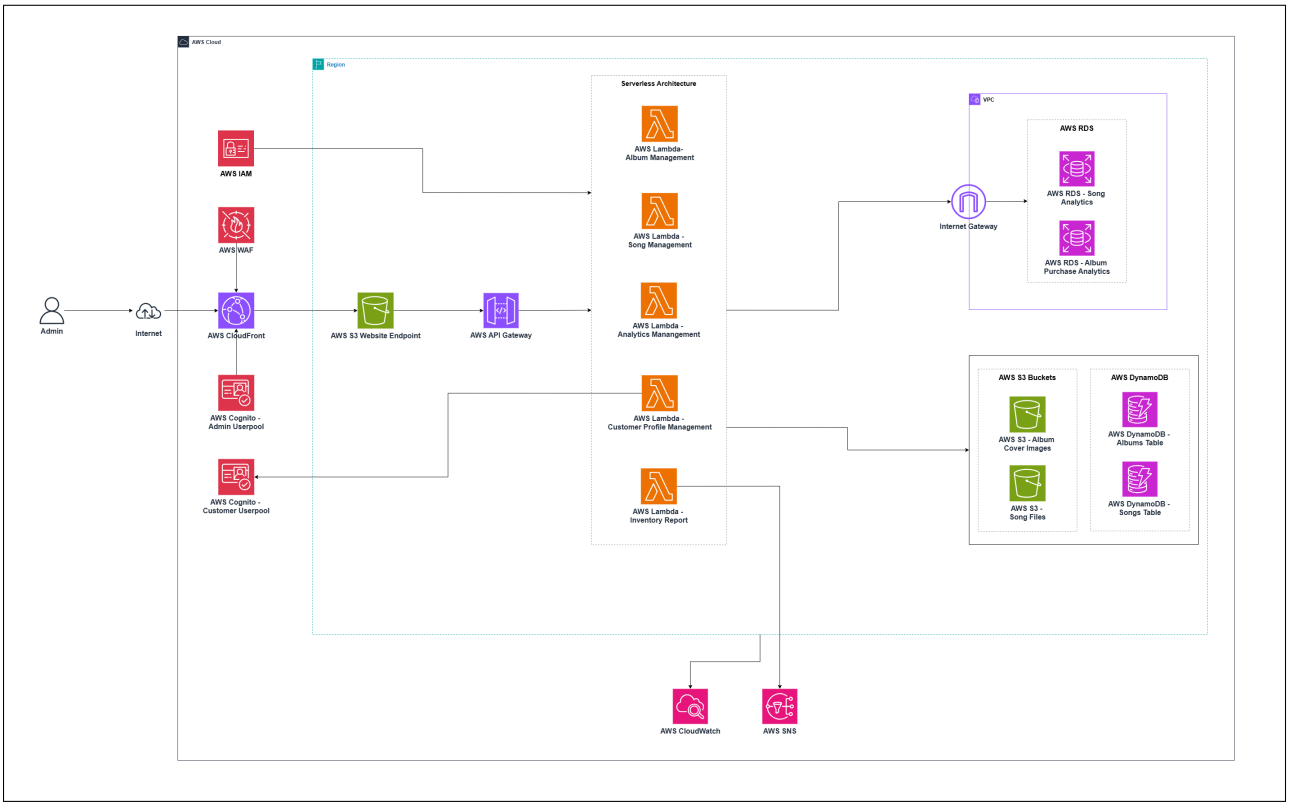


Figure 2.3: AWS Solution Design Diagram for Admin Portal (Refer to figure A.2 for enlarged diagram)

Workflow and Solution Design Explanation - Admin Portal

Component	Service	Function	Connections	Purpose
Admin Access	CloudFront	Distributes admin interface globally	Connected to admin Cognito user pool, AWS WAF, and S3 website endpoint	Provides global low-latency access to the admin website, handles security via WAF, and admin authentication via Cognito

Admin Authentication	Cognito (Admin Pool)	Handles admin login	Connected to CloudFront	Ensures only authorized admin users can access the admin portal
Security	AWS WAF	Protects from web-based threats	Integrated with CloudFront	Filters malicious traffic and secures the admin website
Static Hosting	S3 (Admin Website)	Hosts static admin web interface	Connected to CloudFront and API Gateway	Stores and serves the admin interface (HTML, CSS, JavaScript)
Serverless Architecture	Lambda Functions	Manages backend logic (admin-specific operations)	Connected to API Gateway, IAM, S3 Buckets, RDS, SNS, and customer Cognito user pool	Handles the admin functionality such as managing users, fetching inventory, and interacting with database services
Data Storage (Inventory)	RDS	Stores inventory details (albums and songs)	Connected to Lambda Functions and IAM	Contains detailed album and song information, accessible via lambda functions
File Storage	S3 (Media Storage)	Stores album cover images and song audio files	Connected to lambda Functions and API Gateway	Provides storage for album and song related media files used by lambda functions
Notification Service	SNS	Sends notifications for inventory changes	Connected to Lambda Functions	Sends notifications of inventory when requested

API Gateway	API Gateway	Routes admin requests to backend services	Connected to CloudFront, lambda Functions, and S3 Buckets	Facilitates communication between the admin website and the backend services
IAM Roles	IAM	Manages permissions for Lambda functions	Connected to Lambda Functions, S3 Buckets, and RDS	Ensures that Lambda functions can securely access RDS, S3, and other AWS resources
Monitoring	CloudWatch	Tracks service health and logs	Connected to all services in the AWS Region	Monitors the performance and errors of all AWS services related to the admin operations

Table 2.4: Explanation of the workflow and solution design

2.3 Database Design

The database design for DreamStreamer involves four distinct databases utilizing both NoSQL and relational models to handle the various aspects of the system.

There are two NoSQL databases, powered by DynamoDB, for storing and managing the data related to the albums and songs available on DreamStreamer.

Additionally, there are two relational databases, hosted on AWS RDS, for analytics purposes. One for keeping track of popular songs based on engagement count based on which songs are clicked on the most, and the other for monitoring album sales, thus providing insights on which albums are purchased the most.

2.3.1 Data Models of DynamoDB Tables

Data Model of Album Table

Table attributes	**Type**	**Description**
Album ID (Partition Key)	String	Represent the unique ID of an album
Album Name	String	Represents the name of an album
Artist Name	String	Represents the name of the artist(s) who composed the album
Genre	String	Represents the genre of the album
Track Label	String	Represents the music company which published the album
Band Composition	String	Represents the musical instruments used to compose the songs in the album
Release Date	String	Represents date at which the album was released
Album Art URL	String	Represents the URL to the image of album cover stored in a S3 bucket
Created At	String	Represents the date and time at which the album details were first inserted into the database
Updated At	String	Represents the date and time at which the album was last updated at

Table 2.5: Data model of table used to store album details

Data Model of Song Table

Table attributes	**Type**	**Description**
Song ID (Partition Key)	String	Represents the unique ID of a song
Song Name	String	Represents the name of a song
Album ID	String	Represents the unique ID of the album to which the song is associated with
Album Name	String	Represents the name of the album to which the song is associated with
Artist Name	String	Represents the name of the artist(s) who composed the song
Genre	String	Represents the genre of the song
Track Length	String	Represents the length of the song (in seconds)
Release Date	String	Represents the date in which the song was released
Song File URL	String	Represents the URL of the audio file stored in theS3 bucket
Created At	String	Represents the date and time at which the song details were inserted into the database
Updated At	String	Represents the date and time at which the song details were last updated at

Table 2.6: Data model of table used to store song details

2.3.2 Data Models of RDS Tables

Table attributes	**Type**	**Description**
ID (Primary Key)	String	Represents the unique ID of the record
Album ID	String	Represents the unique ID of the album
Album Name	String	Represents the name of the album to which the song is associated with
Song ID	String	Represents the unique ID of the song
Song Name	String	Represents the name of the song
Artist Name	String	Represents the name of the artist who composed the song
Genre	String	Represents the genre of the song
Track Length	Integer	Represents the length of the song (in seconds)
User Engagement	Integer	Represents the user engagement count

Table 2.7: Data model of table used to store analytcis for popular songs

Table attributes	**Type**	**Description**
ID (Primary Key)	String	Represents the unique ID of the record
Album ID	String	Represents the unique ID of the album
Album Name	String	Represents the name of the album to which the song is associated with
Purchase Count	Integer	Represents the purchase count which is used to track the most purchase albums

Table 2.8: Data model of table used to store album purchase analytics

2.3.3 Justification for Using DynamoDB

DynamoDB was chosen as the database solution for managing album and song data on DreamStreamer for the following key reasons:

- **Scalability:** DreamStreamer needs to handle large volumes of music-related data, such as album details, tracks, and artists. DynamoDB, being a fully managed NoSQL database, can seamlessly scale to meet the demands of high traffic and increasing data without manual intervention.

- **Flexible Data Model:** The data structure for albums and songs may evolve over time as Dreamstreamer introduces new features. DynamoDB's schema-less design allows for flexible data models, enabling changes without requiring complex migrations.

- **High-Throughput Performance:** Given the need for fast and responsive user experiences on both the customer and admin portals, DynamoDB's high throughput and low-latency operations ensure that database queries, even under heavy load, perform efficiently.

- **Integration with AWS Services:** DynamoDB integrates smoothly with other AWS services used in the DreamStreamer architecture, such as AWS Lambda and API Gateway, allowing for efficient serverless interactions between the database and the front-end.

2.3.4 Justification for Using AWS RDS

AWS RDS was chosen as the database solution for storing DreamStreamer's analytics data for the following key reasons:

- **Structured Data Requirements:** The analytics databases store structured data, such as song engagement metrics and album sales figures, which are well-suited to a

relational database model. RDS provides a reliable platform for handling these types of structured, transactional data.

- **Complex Querying:** Analytics involve running complex queries to generate reports, such as identifying the most popular songs or frequently purchased albums. RDS supports advanced SQL querying, enabling efficient data retrieval for detailed insights.

- **Data Integrity:** Maintaining consistency and integrity in the analytics data is critical. RDS offers ACID-compliant transactions, ensuring that the data remains accurate and reliable, especially when tracking real-time user interactions and purchases.

- **Seamless Integration with Other AWS Services:** RDS integrates easily with AWS Lambda, VPC, and other services, making it a natural fit within DreamStreamer's AWS ecosystem. This allows for secure and efficient data handling within the cloud infrastructure.

2.4 Usage of AWS S3

In the solution design S3 buckets were used for:

- Hosting the static websites: customer facing website and admin portal website

- Storing data objects such as album cover images and audio files

2.4.1 Justification of Using S3 Buckets for hosting Static Websites

- **Scalability and Performance:** AWS S3 provides high availability and scalability, allowing the static websites to handle varying levels of traffic without performance degradation. As user demand grows, S3 can automatically scale to accommodate the load.

- **Cost-Effectiveness:** Hosting static websites on S3 is often more cost-effective compared to traditional web hosting services as there are no server maintenance costs, and AWS only charges for the storage and data transfer used, making it ideal for small to medium-sized websites.

- **Easy Deployment and Management:** S3 offers a straightforward interface for deploying static content. Updating the websites involves simply uploading new files to the respective buckets, streamlining the content management process.

- **Separate Buckets for Separation of Concerns:** By hosting the customer-facing website and the admin portal in separate S3 buckets, we maintain a clear separation of concerns. This not only simplifies management and organization but also enhances security by limiting access to the admin portal to authorized personnel only.

2.4.2 Justification of Using S3 Buckets for Storing Images and Audio Files

- **Durability and Availability:** AWS S3 provides 99.9% durability, ensuring that data such as album cover images and audio files are stored reliably.

- **Scalable Storage:** S3 can scale to accommodate any amount of data, making it suitable for storing potentially large collections of audio files and images without worrying about running out of space.

- **Cost-Effectiveness:** Storing images and audio files in S3 is cost-efficient, especially for large datasets.

- **Optimized for Media Delivery:** S3 integrates seamlessly with AWS services like CloudFront for content delivery, allowing for fast loading times for media files. This is essential for providing a smooth user experience on the customer-facing website.

- **Separation of Buckets for Organization and Access Control:** Using two separate buckets for images and audio files allows for better organization of assets and simplified access management. This approach ensures that permissions can be finely tuned based on the type of media, enhancing security. For instance, access to audio files can be restricted to specific applications or users, while images can be made publicly accessible for display on the website.

- **Compliance and Data Management:** Storing images and audio files in separate buckets allows for easier compliance with data management policies. For instance, if regulations require specific data retention periods or access controls, having separate buckets simplifies tracking and enforcement.

Chapter 3

Implementation

3.1 AWS API Gateway

AWS API Gateway is used for the frontend to communicate with the backend databases, like AWS DynamoDB, AWS RDS and S3 buckets, by integrating the APIs with lambda functions.

Resource	**HTPP Method**	**API Endpoint**	**Integrated Lambda Function**
Albums	POST	https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/add-album	addAlbum
	GET	https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/get-album	getAlbumDetails
	GET	https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/get-album-by-id	getAlbumDetailsById
	GET	https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/get-album-cover	getAlbumCover
	PUT	https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/update-album	updateAlbum

	POST	https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/upload-new-album-cover	uplaodNewAlbumCover
	POST	https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/delete-album	deleteAlbum
Songs	POST	https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/add-song	addSong
	GET	https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/get-song	getSongDetails
	GET	https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/get-song-by-id	getSongDetailsById
	GET	https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/get-songs-for-album	getSongForAlbum
	PUT	https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/update-song	updateSong
	POST	https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/upload-new-song	uploadNewSong
	POST	https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/delete-song	deleteSong
Analytics	POST	https://dfg5xvb41m.execute-api.eu-west-1.amazonaws.com/dev/update-engagement	addAnalytics
	POST	https://dfg5xvb41m.execute-api.eu-west-1.amazonaws.com/dev/update-price-analytics	addPriceAnalytics
	GET	https://dfg5xvb41m.execute-api.eu-west-1.amazonaws.com/dev/get-analytics	getAnalytics

	GET	https://dfg5xvb41m.execute-api.eu-west-1.amazonaws.com/dev/get-price-analytics	getPriceAnalytics
Report	POST	https://25sjzr7wk2.execute-api.eu-west-1.amazonaws.com/dev/request-inventory-report	sendInventoryReport
Profile	GET	https://enltfklsz7.execute-api.eu-west-1.amazonaws.com/dev/get-customer-profile	getCustomerProfiles
	DELETE	https://enltfklsz7.execute-api.eu-west-1.amazonaws.com/dev/delete-customer-profile	deleteCustomerProfile

Table 3.1: List of all API endpoints in the solution along with its integrated lambda function

3.2 Lambda Functions

Lambda Function	Description
addAlbum	This function handles adding a new album to a DynamoDB table named 'Albums' and uploading the album's cover art to an S3 bucket. A pre-signed URL is generated to allow the client to upload the album cover to the S3 bucket 'dreamstreamer-album-art'. The album data, along with the S3 URL of the album cover, is then saved to DynamoDB. If successful, the function responds with a success message and the pre-signed URL for the image upload. If there is an error, it returns an appropriate error message.
getAlbumDetails	Fetches a list of all albums from the DynamoDB table 'Albums'. It scans the table and returns all the album details stored.
getAlbumCover	Retrieves the album cover URL for a specific album using the albumId.
getAlbumDetailsById	Fetches detailed information of a specific album based on its albumId from the DynamoDB 'Albums' table. This function is used in the admin portal for when an album is selected, the corresponding album fields like name, genre, artist, track label, are automatically populated for updating purposes.
updateAlbum	Updates the details of an existing album. If a new album cover image is uploaded, it updates the album cover URL too.
uploadNewAlbumCover	Generates a pre-signed URL for uploading a new album cover image to an S3 bucket. This is used during the album update process to upload the album cover and return the file URL, which is stored in the database.

deleteAlbum	Deletes an album from the Albums table and also deletes associated songs from the Songs table using a Global Secondary Index (GSI); 'albumId-songId-index'. The GSI retrieves all songs associated with an album ID from the 'Songs' table and then delete them. Additionally, deletes the album cover from S3.
addSong	Adds a new song to the Songs table, associated with an album. It first queries the Albums table by albumName to retrieve the albumId. It also generates a pre-signed URL for uploading the song file to S3.
getSongDetails	Fetches a list of all songs from the DynamoDB table 'Songs'. It scans the table and returns all the song details stored.
getSongForAlbum	Retrieves songs under a specific album using a GSI; 'albumName-songId-index' based on the albumName.
getSongDetailsById	Retrieves song details by songId from the 'Songs' table. Used in the admin portal to auto-fill song details when selecting a song for update.
updateSong	Updates an existing song's metadata in a DynamoDB table 'Songs'. If a new song file URL is provided, it updates that as well.
uploadNewSong	Generates a pre-signed URL for uploading a new song file to an S3 bucket 'dreamstreamer-song'.
deleteSong	Deletes a song entry from DynamoDB and also deletes the corresponding song file from the S3 bucket.
addAnalytics	Tracks user engagement for songs by adding/updating engagement count in an RDS database. If a song exists, it increments the engagement count; otherwise, it adds the song entry.

addPriceAnalytics	Tracks the number of purchases of albums in an RDS database. It updates the purchase count for existing albums or adds a new record if the album doesn't exist.
getAnalytics	Fetches song engagement data from the RDS database based on filters such as by artist, album, or genre, and sorts it by engagement count.
getPriceAnalytics	Retrieves purchase data from the RDS database, allowing for sorting by album name or purchase count.
sendInventoryReport	Queries DynamoDB for all album and song data, then generates and sends an inventory report via SNS to a pre-determined email address.
getCustomerProfile	Fetches a list of users from an AWS Cognito customer user pool. It retrieves up to 60 users at a time, paginating if necessary, and returns the complete list of users.
deleteCustomerProfile	Deletes a user from the Cognito customer user pool based on their username.
customerSignup-EmailAuth	Acts as a pre-signup trigger in AWS Cognito to ensure no duplicate email signups. Before a user is signed up, this Lambda checks if the email is already associated with another user. If a duplicate is found, it throws an error to prevent the signup.

Table 3.2: Description of all lambda functions used in the implementation

3.3 Frontend Development

The frontend of the DreamStreamer application was developed using the following technologies:

- React framework
- Tailwind CSS
- HTML
- CSS
- JavaScript

3.4 Dynamic Data Fetching and Displaying

To ensure the website always displays the latest information, API calls are used to dynamically retrieve data from the backend. The APIs are integrated with AWS Lambda functions, which interact with the databases and send the data to the frontend for display.

3.4.1 Overview of API and Lambda Integration Flow for Data Fetching and Displaying

- The front-end uses JS fetch API to make requests to AWS API Gateway, which triggers AWS lambda functions. These functions handle the retrieval of data from AWS DynamoDB and RDS. For example, when a user views a specific song or album, the Lambda function fetches the relevant data such as album details, songs etc from the databases.
- For album and song details, DynamoDB is queried via lambda, while RDS is used for analytics data like song popularity or purchase statistics. The API call triggers a

lambda function that fetches and returns the data as a JSON response. The React useEffect hook is used to call these APIs when the component mounts, ensuring that the most recent data is fetched when the page loads or updates.

- Once the data is fetched, it is stored in the React component's state using useState and rendered dynamically. This allows the website to update its content without reloading the entire page. Users can filter albums by genre or artist, and based on their input, new API calls are made, updating the displayed content dynamically.

3.4.2 Implementation of Dynamic Data Fetching and Display in the Customer Facing Website

Fetching Album Data for The Frontend

- In the AlbumDetail component, an empty array is initialized using the useState hook to store album data (figure D.1).
- The useEffect hook is used to make an API call using fetch to retrieve album details from the DynamoDB "Albums" table (figure D.2).
- The API call triggers a Lambda function that retrieves album data from the DynamoDB table and returns it in JSON format.
- The JSON response is stored in state and dynamically rendered in the frontend as a list of albums.
- The Lambda function uses the AWS SDK to perform a scan operation on the "Albums" table (figure D.3).
- The result is returned in JSON format to the frontend via API Gateway.
- The map function is used to dynamically display each album as a card where each card includes details of an album (figure D.4). See figure B.8 for rendered UI.

Fetching Song Data for The Frontend

- In the SongDetail component, an empty array is initialized using the useState hook to store song data (figure D.5).

- The useEffect hook is used to make an API call using fetch to retrieve song details from the DynamoDB "Songs" table (figure D.5).

- The API call triggers a lambda function that retrieves song data from the DynamoDB table and returns it in JSON format.

- The JSON response is stored in state and dynamically rendered in the frontend as a list of song.

- The Lambda function uses the AWS SDK to perform a scan operation on the "Songs" table (figure D.5).

- The result is returned in JSON format to the frontend via API Gateway.

- The map function is used to dynamically display each album as a card where each block includes details of a song (figure D.5). See figure B.10 for rendered UI.

3.5 CRUD Operations

The CRUD operations are mostly performed in the admin portal on albums and songs data. Also, analytics data in AWS RDS in analyzed in the admin portal. Furthermore, CRUD operations are also performed on album cover images and song files which are stored in S3 buckets.

The read operation in the admin portal functions similar to how the read operation functions in the customer facing website.

3.5.1 Overview of API and Lambda Integration Flow for CRUD Operations

- The front-end employs the JS fetch API to make HTTP requests to AWS API Gateway, triggering specific AWS Lambda functions to performCRUD operations on album and song data.

- **Create Operation:** When a user submits a form to add a new album, the front-end sends an HTTP POST request to the API Gateway, which invokes a Lambda function. The Lambda function generates a unique album ID and stores the album's metadata such as album name, genre, artist, and release year in DynamoDB, and returns a pre-signed URL for securely uploading album artwork to S3. After the image is uploaded, the album's artwork URL is stored alongside the album data in DynamoDB.

- **Read Operation:** To fetch albums or songs, the front-end calls the API using fetch API, via React's useEffect hook for fetching data on component mount. The lambda function retrieves data from DynamoDB based on the query parameters like album ID or filters such as genre or artist. The retrieved data is returned in JSON format and rendered dynamically using React's useState to update the page without a full reload.

- **Update Operation:** When users edit an album's details or update album artwork, the front-end triggers an HTTP PUT request to the API Gateway. A corresponding Lambda function updates the relevant fields in DynamoDB including the album's metadata and a new S3 pre-signed URL if the artwork is updated. The Lambda function updates the album's metadata, including timestamps, and returns a success response once the update is completed.

- **Delete Operation:** For album or song deletion, the front-end sends an HTTP POST request containing the album or song ID to the API Gateway. The Lambda function deletes the record from DynamoDB and any associated media files like album artwork

or song files from S3. For albums, it also triggers the deletion of all associated songs from the DynamoDB table and their corresponding audio files from S3.

- **S3 Integration:** The lambda functions use AWS S3 to manage file uploads like album cover image or song files by generating pre-signed URLs, which allow the front-end to securely upload media files. S3 is also used to delete these media files when the corresponding album or song is deleted from DynamoDB.

- **State Management and Rendering:** Once the CRUD operations are successfully completed, the React components update the displayed data in real time. Operations like adding a new album or deleting an existing one trigger state changes using useState, ensuring the page content is updated dynamically without needing a full page reload. This allows for a seamless user experience when managing albums and songs on the platform.

3.5.2 Implementation of CRUD Operations in the Admin Portal

Creating a New Album

- In the AddAlbum component, useState is used to track album form data and album art file (figure E.1).

- On form submission, a POST request is sent to API Gateway, passing album details and file metadata (figure E.2-E.3).

- The lambda function receives the album data and generates a presigned URL for uploading the album art to an S3 bucket.

- After receiving a presigned URL from the lambda function, the file is uploaded directly to S3 via a PUT request. Then, the album details including the S3 file URL are then sent to DynamoDB via the lambda function (figure E.4- E.5).

Fetching Album Data for The Frontend

- In the frontend an empty array is initialized using the useState hook to store album data (figure D.1).
- The useEffect hook is used to make an API call using fetch to retrieve album details from the DynamoDB "Albums" table (figure D.2).
- The API call triggers a Lambda function that retrieves album data from the DynamoDB table and returns it in JSON format.
- The JSON response is stored in state and dynamically rendered in the frontend as a list of albums.
- The lambda function uses the AWS SDK to perform a scan operation on the "Albums" table (figure D.3).
- The result is returned in JSON format to the frontend via API Gateway.
- The map function is used to dynamically display each album as a card where each card includes details of an album (figure D.4). See figure B.8 for rendered UI.

Updating an Album

- Displays album data in a form for editing, allowing users to update fields or change the album art (figure E.6).
- If album art is updated, a new presigned URL is requested, and the updated file is uploaded to S3.
- Sends a PUT request with the updated album details (including the new S3 URL if the file was changed) to the API Gateway (figure E.7-E.8).
- The lambda function receives the updated album details and updates the existing album entry in DynamoDB.

- If new album art is provided, the S3 URL is updated in the "Albums" table (figure E.9).

Deleting an Album

- When deleting an album, the front-end sends an HTTP POST request to the API Gateway. The request contains the album ID and the URL of the album artwork stored in S3 (figure E.10).

- Then the lambda function handles the deletion by deleting the album's metadata from the "Albums" table in DynamoDB by using the album's ID as the key.

- The function also queries the Songs table to find all songs associated with the album via the albumId. It deletes each song record from DynamoDB using batch processing.

- The function extracts the file name of the album artwork from the provided albumArtUrl and deletes the corresponding image file from S3 (figure E.11-E.12).

- The front-end updates the album list by filtering out the deleted album in the component's state, ensuring the UI reflects the deletion without needing a page reload.

The CRUD operations for "Songs" also follows a similar structure to that of "Albums". (Refer Lambda Functions chapter in appendix for code of Songs CRUD functions)

3.6 Filtering Data

- useState is used to track the list of all songs and the list of filtered songs based on user input such as album, genre, or artist (figure F.1).

- When the component loads, a useEffect hook fetches all the songs from an API and the fetched song data is stored in the songs state (figure F.2).

- Another useEffect hook is used to watch for changes in the filter state and based on the selected filters, the filteredSongs array is updated using the .filter() function to match the selected album, genre, or artist (figure F.3).
- When a user selects a new value for album, genre, or artist, the handleFilterChange function updates the respective filter's state (figure F.4).
- To populate the filter dropdowns, unique values of albums, genres, and artists are derived from the songs array using Set to remove duplicates.
-
- For the filter inputs in the dropdown menus for each filter (album, genre, artist) only the unique values generated earlier are rendered and binded in each dropdown to the respective filter state (figure F.5).
- The filteredSongs array is rendered in a list or card format, displaying only the songs that match the selected filters (figure F.6-F.7).

3.7 Analytics Operations

Adding Popular Song Analytics

- When the user clicks on a song the handleSongClick function uploads a payload containing song details through the API Gateway to the lambda function (figure G.1).
- The lambda function processes incoming events from user interactions (e.g., song clicks).
- Then a query checks if the song already has an entry in the database.
- If it exists, the user_engagement count is incremented; if not, a new record is inserted with an initial engagement of 1.

- The function returns a JSON response indicating success or failure, with appropriate status codes (figure **??-??**).

Adding Album Purchase Analytics

- When a user clicks on the purchase button on an album the handlePurchaseClick function constructs a payload with album details and sends it to the "update-price-analytics" API (figure G.4).

- The lambda function processes incoming requests, extracting album details from the event body. A query checks if the album already has an entry in the database.

- If it exists, the purchase count is incremented; if not, a new record is inserted with an initial count of 1.

- The function returns a JSON response indicating success or failure, with appropriate status codes (figure G.5-G.6).

Retrieving Popular Song Analytics

- The frontend fetches song engagement data through an API endpoint. The API call includes a filter by artist, album, or genre, that modifies the query based on user selection (figure G.7).

- The lambda function processes the filter query parameter and constructs SQL queries dynamically based on the filter and then retrieves song engagement data from the "song_engagement" table.

- Song data is displayed in a table format, showing song ID, album ID, song name, album name, genre, artist name, and user engagement metrics.

- The list updates dynamically when the user selects a different filter from the dropdown menu.

- If there is an error during data retrieval, the backend returns a 500 status code with an error message (figure G.8-G.9).

Retrieving Purchase Analytics

- The frontend fetches purchase analytics data via an API endpoint. It allows filtering by album name in ascending or descending order (figure G.10).
- The lambda function receives a filter query parameter and constructs SQL queries accordingly. If no filter is specified, it defaults to ordering by purchase count in descending order.
- The data is displayed in a structured format, showing album ID, album name, and purchase count, updating dynamically based on the selected filter.
- If data retrieval fails, the backend responds with a 500 status code and an error message (figure G.11-G.12).

3.8 Configurations of AWS S3 and AWS API Gateway

To ensure seamless communication between the frontend, lambda functions, and S3 buckets, specific configurations are required. First, an appropriate S3 bucket policy is set, granting the lambda functions necessary permissions to access the buckets. Additionally, CORS (Cross-Origin Resource Sharing) is configured on the S3 buckets to allow secure requests from the frontend.

On the AWS API Gateway side, CORS is also configured to enable the frontend to interact with the lambda functions via API Gateway. This ensures that the frontend can send requests and receive responses without cross-origin issues. These configurations allow the lambda functions to access the S3 buckets and the frontend to securely communicate with the backend services.

The following images showcase the policies and permissions that were set.

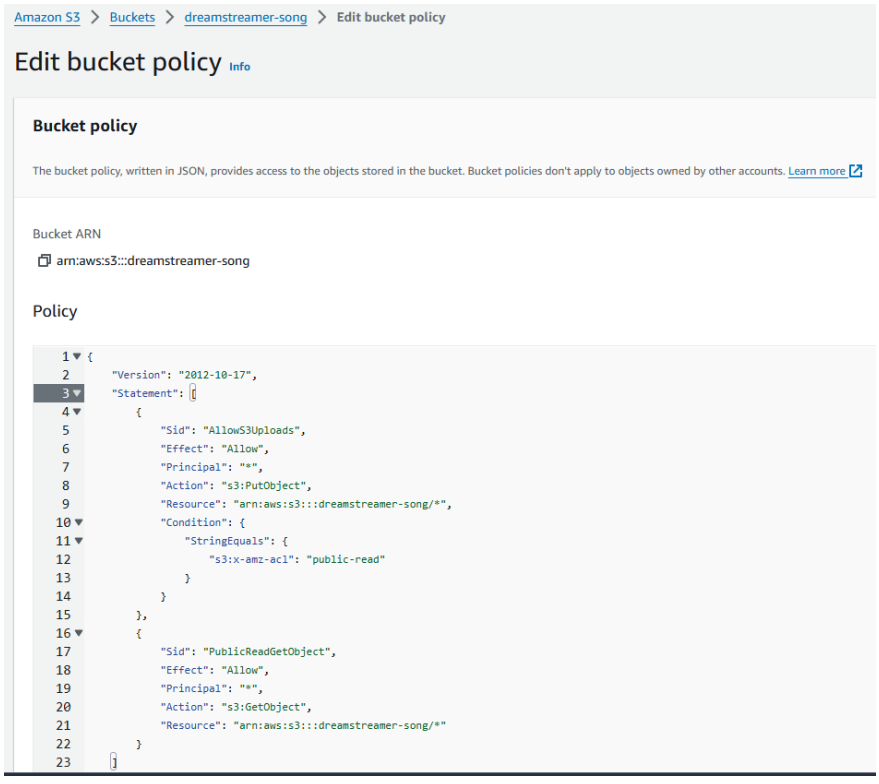


Figure 3.1: Policies set for buckets that store album cover images and song audio files

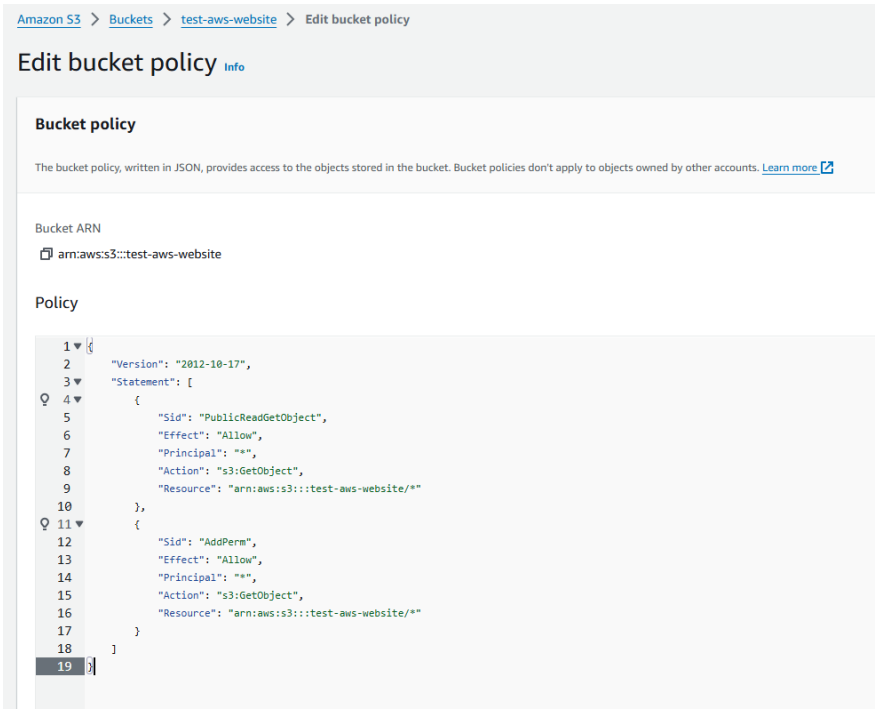


Figure 3.2: Policies set for buckets host the static websites

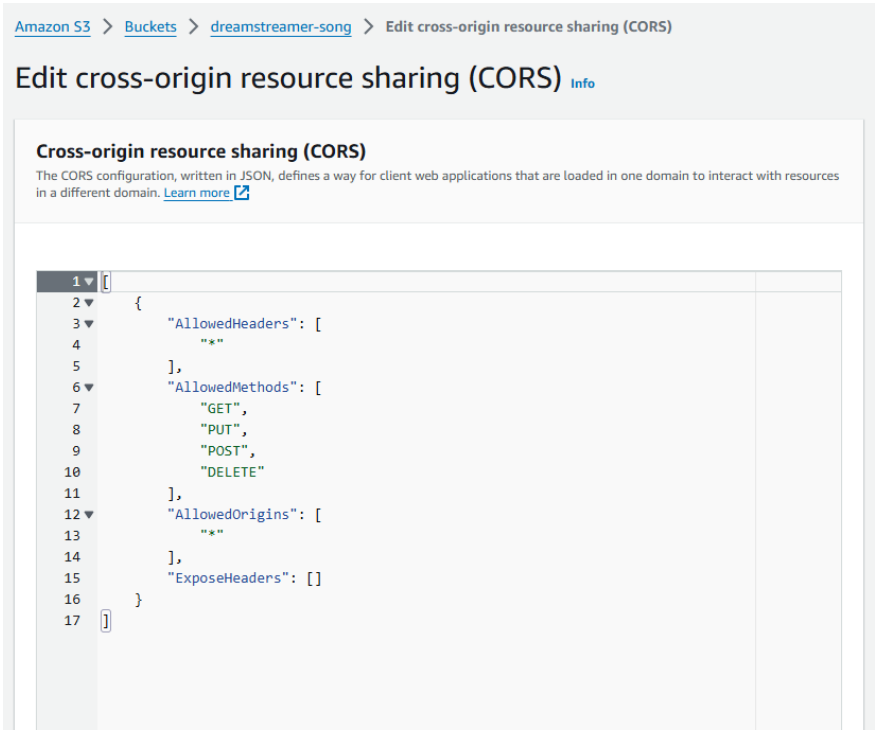


Figure 3.3: CORS policies set for all buckets

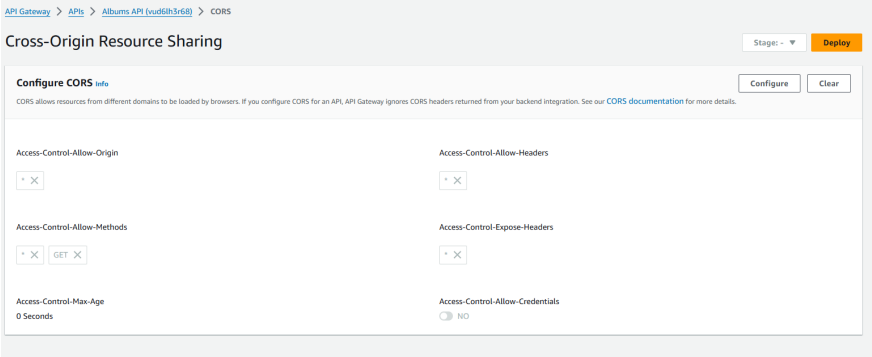


Figure 3.4: CORS policies set for APIs

Chapter 4

Authentication and Security

4.1 AWS Cognito

4.1.1 Cognito Userpool Setup for Customers

For customers, the cognito user sign-in option is set as "user name" so that users can login into the application using their username and passwords as login credentials.
The other required attributes that are included in this userpool are:

- Address
- Name
- Phone number
- Custom:user_type

4.1.2 Cognito Userpool Setup for Admins

For admin users, a seperate userpool is created and the cognito user sign-in option is set as "email" so that admin users can login into the application using their email address and passwords as login credentials.

4.2 Cognito Signup Process

4.2.1 Customer Signup Process

- The CognitoUserPool is initialized using the UserPoolId and ClientId provided from AWS Cognito.
- The signup form collects details like username, name, email, password, address, and phone number from the user.
- Custom user attributes such as custom:user$_typearealsocapturedduringthesignupprocess.Attributesl$

- The userPool.signUp() method is used to register the user in the Cognito User Pool. This method sends all user attributes to Cognito.
- If the signup is successful, a confirmation code is sent to the user's email.
- After receiving the confirmation code via email the user can input it into the form.
- The cognitoUser.confirmRegistration() method is used to confirm the user's signup with the code provided.
- Error handling is implemented for common issues such as:
  - Username already exists (UsernameExistsException).
  - Invalid email or other parameters (InvalidParameterException).

4.3 Cognito Login Process

4.3.1 Customer Login Process

- Similar to the signup process, the CognitoUserPool is initialized using the UserPoolId and ClientId.

- The login form collects the username and password from the user.
- The AuthenticationDetails class is used to create an authentication object containing the username and password.
- The CognitoUser object is initialized with the user's username.
- The cognitoUser.authenticateUser() method is used to authenticate the user by passing in the AuthenticationDetails.
- On successful authentication, an accessToken is returned, representing the user's session.
- The user's access token is logged, and the AuthContext is updated with the authenticated user data.
- Upon successful login, the user is redirected to the "Songs" page.
- If authentication fails (e.g., wrong password or unverified account), an error message is displayed, and the user is notified.

4.3.2 Admin Login Process

- When the admin submits their email and password through the login form, the handleSubmit function is triggered.
- The login function from AuthContext is called, passing the admin's email and password. This function is responsible for authenticating the user against the Cognito User Pool.
- In the AuthContext.js file, login creates a new CognitoUser object using the provided email and password. This user is then authenticated using Cognito's authenticateUser method, which either succeeds or fails based on the credentials.

- If the authentication is successful, the session is established, and the authenticated user information is saved in the user state of the AuthContext.
- If the authentication fails, an error message is passed to the callback, which updates the error state in Login.js and displays the error on the form.
- Once logged in, if authentication is successful, the navigate function is called to redirect the admin to the homepage (or another admin-specific route, if desired).

4.4 AuthContext Overview

The AuthContext handles session persistence by checking if a user is already logged in by using UserPool.getCurrentUser(). If a user is found, their session is validated, and the users attributes fetched and stored in the context.

4.5 Explanation of ProtectedRoute Component

The ProtectedRoute component is responsible for restricting access to certain routes based on the user's authentication status (figure H.11). It works by accessing the AuthContext and checking whether there is an authenticated user. If the user is not authenticated it redirects to the login page, otherwise if the user is authenticated the child components are rendered and the user is allowed to access them.

This component is important because:

- **Security:** ProtectedRoute ensures that only authenticated users can access certain parts of the application. For example, pages meant for admin users or sensitive content that should not be publicly available are shielded from unauthorized access.
- **Prevents Unauthorized Access:** Unauthorized users (those who haven't logged in) are automatically redirected to the login page when they attempt to access a protected

route. This keeps sensitive or admin-only pages inaccessible until the user provides valid login credentials.

4.5.1 Justification for Using Two Different Userpools

- **Security and Access Control:** By having separate user pools stricter access controls can be implemented for each user type. For instance, admins often need elevated permissions and access to sensitive data, while customers require limited access. Keeping them in different pools helps enforce these boundaries more effectively.

- **Minimized Risk:** If a security breach occurs, isolating user pools minimizes the risk of exposure. A compromise in the customer user pool would not directly affect admin accounts and vice versa.

- **Easier User Management:** Different user pools allow for clearer management of user attributes and settings. For instance, admin users may need different attributes compared to customer users. This separation can simplify user management and reduce complexity.

- **Flexible Scaling:** Modifying or upgrading the authentication mechanisms for one user type like implementing different security policies for admins can be done easily without affecting the other users. Also, separate user pools provide the flexibility to expand without complicating the existing user management system.

4.6 Identity and Access Management

Identity and Access Management (IAM) roles are a critical component of AWS security as they allow to define permissions that can be assumed by AWS services, applications, or users. This enables secure and controlled access to AWS resources without needing to manage static credentials.

In this solution, IAM roles are primarily used to create roles that are attached to AWS lambda functions. By assigning these roles to lambda functions, permissions are granted for the lambda function to access other resources like S3, DynamoDB, or API Gateway. This setup ensures that lambda can perform its tasks—such as reading data from a database or writing to a storage bucket while adhering to the principle of least privilege, thus enhancing overall security.

IAM Role	**Policies**	**Description**	**Attached Lambda Function**
lambdaAlbum	AmazonDynamoDB-FullAccess	Provides full access to AWS DynamoDB	addAlbum updateAlbum uploadNew-AlbumCover deleteAlbum
	AmazonS3FullAccess	Provides full access to AWS S3 buckets	
	AmazonAPIGateway-Administrator	Provides full access to create/edit/delete APIs in AWS API Gateway	
lambdaAlbum-ReadAccess	AmazonDynamoDB-ReadOnlyAccess	Provides read only access to Amazon DynamoDB	getAlbumDetails getAlbumCover getAlbumDetails-ById
	AmazonS3ReadOnly-Access	Provides read only access to all AWS S3 buckets	
	AmazonAPIGateway-Administrator	Provides full access to create/edit/delete APIs in AWS API Gateway	

lambdaSong	AmazonDynamoDB-FullAccess	Provides full access to AWS DynamoDB	addSong updateSong uploadNewSong deleteSong
	AmazonS3FullAccess	Provides full access to AWS S3 buckets	
	AmazonAPIGateway-Administrator	Provides full access to create/edit/delete APIs in AWS API Gateway	
lamdaSongRead-Access	AmazonDynamoDB-ReadOnlyAccess	Provides read only access to Amazon DynamoDB	getSongDetails getSongForAlbum getSongDetails ById
	AmazonS3ReadOnly-Access	Provides read only access to all AWS S3 buckets	
	AmazonAPIGateway-Administrator	Provides full access to create/edit/delete APIs in AWS API Gateway	
lambdaInventory-Report	AmazonDynamoDB-ReadOnlyAccess	Provides read only access to Amazon DynamoDB	sendInventoryRepor
	AmazonSNSFullAccess	Provides full access to Amazon SNS	
lambdaRDS-Analytics	AmazonRDSFullAccess	Provides full access to Amazon RDS	addAnalytics addPriceAnalytics
	AmazonAPIGateway-Administrator	Provides full access to create/edit/delete APIs in AWS API Gateway	
lambdaRDS-ReadAnalytics	AmazonRDSRead-OnlyAccess	Provides read only access to Amazon RDS	getAnalytics getPriceAnalytics
	AmazonAPIGateway-Administrator	Provides full access to create/edit/delete APIs in AWS API Gateway	
lambdaGet-CustomerProfile	AmazonCognito-ReadOnly	Provides read only access to Amazon Cognito resources	getCustomer-Profile
	AmazonAPI-GatewayAdministrator	Provides full access to create/edit/delete APIs in AWS API Gateway	

lambdaDelete-CustomerProfile	AmazonCognito-PowerUser	Provides administrative access to existing Amazon Cognito resources. You will need AWS account admin privileges to create new Cognito resources	deleteCustomer-Profile
	AmazonAPIGateway-Administrator	Provides full access to create/edit/delete APIs in AWS API Gateway	

Table 4.1: List of IAM roles used and the associated policies with the lambda functions they are integrated to

4.7 Secure Content Delivery with Cloudfront

4.7.1 Default SSL Certificate for Secure Communication

To ensure that all communications between users and the website are secure, Amazon CloudFront was configured with a default SSL certificate. This setup ensures:

- **End-to-End Encryption:** All data transferred between the end-user and CloudFront is encrypted using HTTPS. This protects sensitive information, such as login credentials and personal data, from being intercepted by malicious actors.

- **Trust and Browser Compatibility:** The default SSL certificate provided by AWS is trusted by major browsers, ensuring that users accessing the website will not encounter security warnings, enhancing trust and confidence in the application.

- **Compliance and Best Practices:** Using SSL/TLS encryption helps meet security compliance standards such as PCI-DSS, HIPAA, and others, ensuring that the application adheres to industry security best practices.

4.7.2 AWS Web Application Firewall (WAF)

AWS WAF (Web Application Firewall) was integrated with CloudFront to protect the application from common web threats and attacks. By applying WAF at the CloudFront distribution level, the following protections are enforced:

- **Protection Against Common Attacks:** WAF helps mitigate vulnerabilities such as SQL injection, cross-site scripting (XSS), and other OWASP top 10 attacks by filtering incoming traffic before it reaches the backend systems.
- **Customizable Security Rules:** Custom security rules can be created to allow, block, or rate-limit requests based on specific patterns, IP addresses, geographic location, and more. This helps to prevent DDoS attacks or any other malicious traffic patterns that could disrupt the application's availability.
- **Real-time Monitoring and Alerts:** WAF integration provides real-time monitoring and logging of any suspicious activities, enabling administrators to take immediate action when necessary.

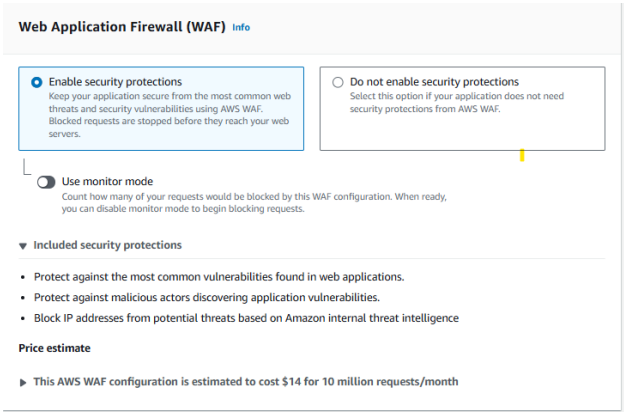


Figure 4.1: AWS WAF settings in Cloudfront integration

Chapter 5

Testing and Validation

5.1 Testing

To ensure that the system operates as intended, both the API and Lambda functions were tested through the following methods:

- **API Testing using Postman:** Postman was used to manually test the API endpoints through various HTTP requests such as GET, POST, PUT, and DELETE and verifying the results of the response. This allowed for testing the communication between the frontend and backend, validating that data was correctly retrieved, created, updated, and deleted as per the expected behavior.

- **Lambda Function Testing in AWS Console:** The AWS Lambda console was utilized to execute and debug the Lambda functions. By passing different input event payloads and observing the function output and logs to confirm that each lambda function performed its functionality correctly, including accessing and interacting with other AWS resources such as S3 and DynamoDB.

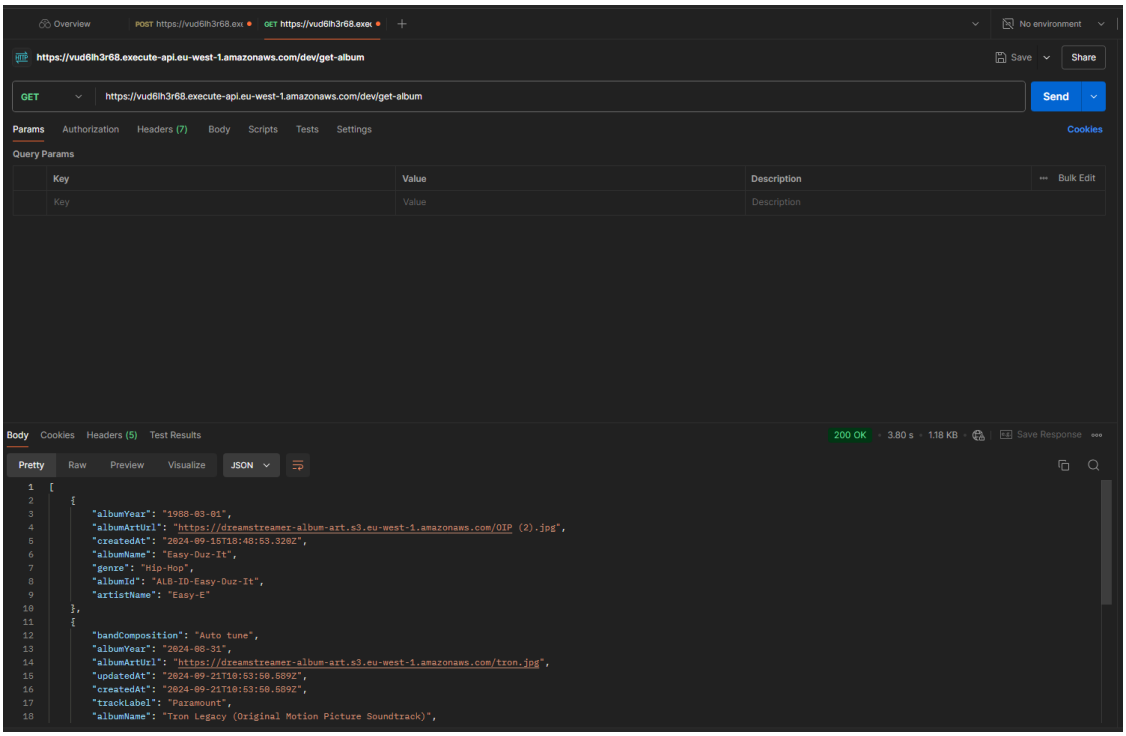


Figure 5.1: Testing the GET API in Postman for retrieving album details

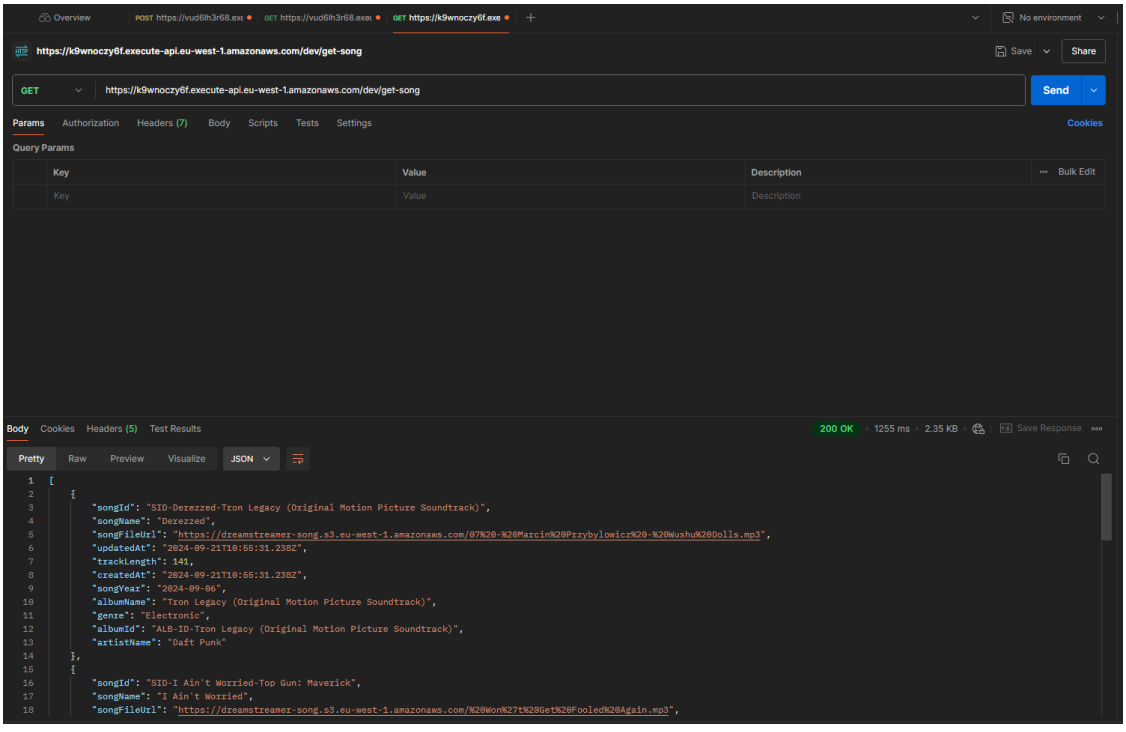


Figure 5.2: Testing the GET API in Postman for retrieving song details

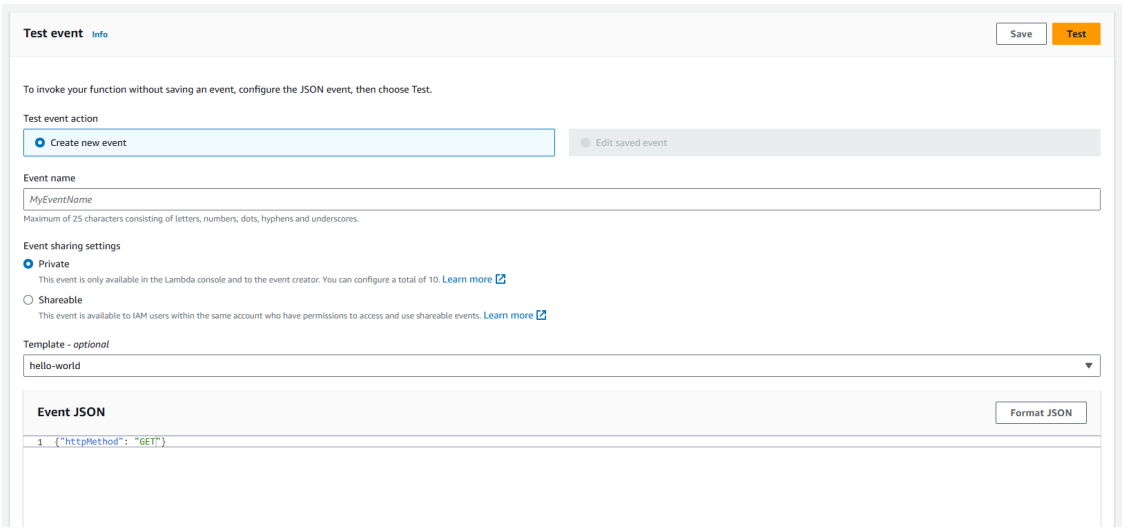


Figure 5.3: The test event for the lambda function in AWS console for retrieving album details

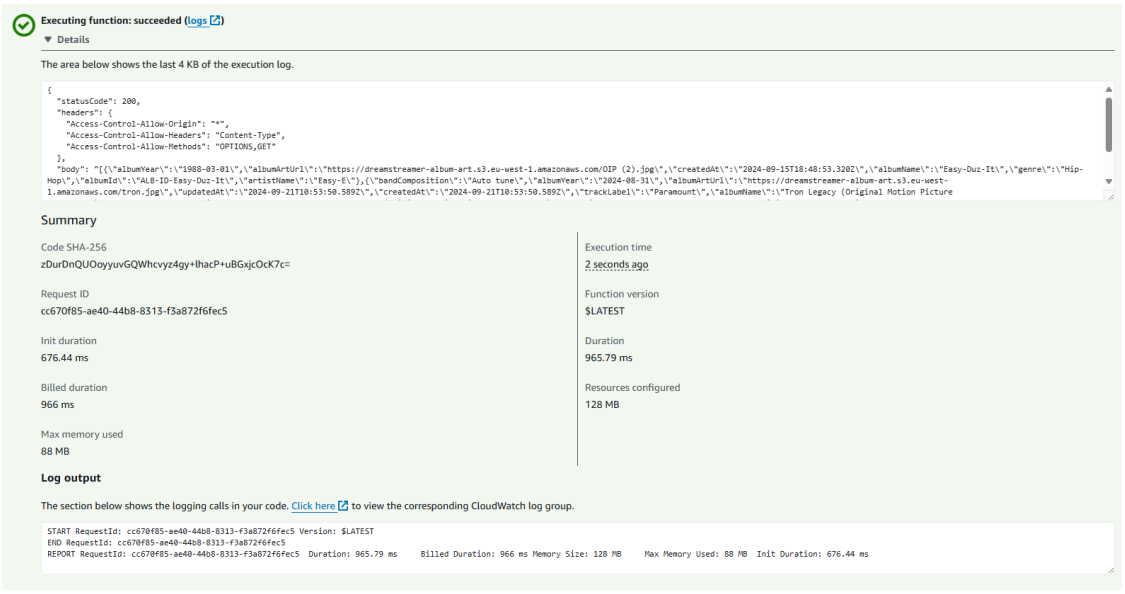


Figure 5.4: The test result for the lambda function in AWS console for retrieving album details

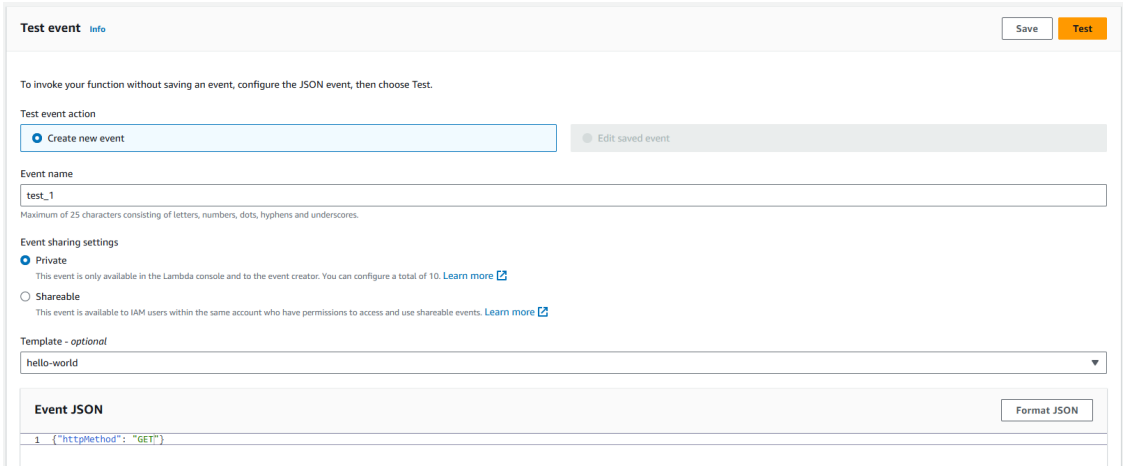


Figure 5.5: The test event for the lambda function in AWS console for retrieving song details

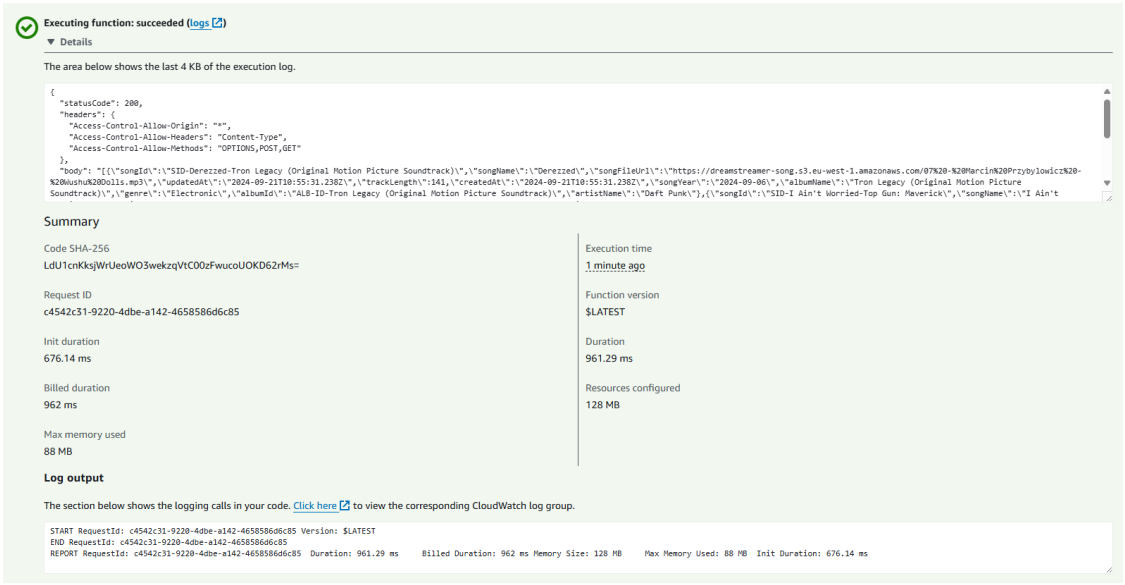


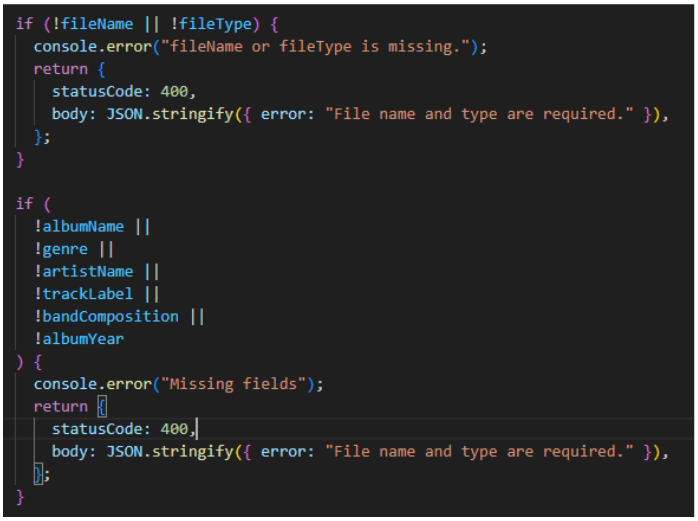
Figure 5.6: The test result for the lambda function in AWS console for retrieving song details

5.2 Validation

The validation process was implemented on the frontend to ensure that all required fields are filled out correctly before submitting the form, enhancing the user experience and preventing incomplete or incorrect data from being sent to the backend. Additionally, validation was also implemented within the Lambda functions to ensure that no fields were left empty or invalid, providing a secondary layer of validation to prevent improper data from being processed at the backend level. This dual-layer validation helps maintain data integrity and ensures smoother backend operations.

```
const handleSubmit = async (e) => {
  e.preventDefault();

  const {
    albumName,
    genre,
    artistName,
    trackLabel,
    bandComposition,
    albumYear,
    albumArtUrl,
  } = formData;

  if (
    !albumName ||
    !genre ||
    !artistName ||
    !trackLabel ||
    !bandComposition ||
    !albumYear ||
    albumArtUrl
  ) {
    alert("Please fill out all the fields before submitting!");
    return;
  }
```

Figure 5.7: Code snippet of form validation in AddAlbum component



```
if (!fileName || !fileType) {
  console.error("fileName or fileType is missing.");
  return {
    statusCode: 400,
    body: JSON.stringify({ error: "File name and type are required." }),
  };
}

if (
  !albumName ||
  !genre ||
  !artistName ||
  !trackLabel ||
  !bandComposition ||
  !albumYear
) {
  console.error("Missing fields");
  return {
    statusCode: 400,
    body: JSON.stringify({ error: "File name and type are required." }),
  };
}
```

Figure 5.8: Code snippet of lambda validation in addAlbum lambda function

Chapter 6

Deployment and Hosting

The website was deployed using AWS S3 for static website hosting. To enhance the performance and security of the website, Amazon CloudFront, a content delivery network (CDN), was integrated. The benefits of using CloudFront include:

- **Global Content Distribution:** CloudFront delivers website content to users from edge locations closest to them, reducing latency and improving load times, especially for a global audience.

- **Security:** CloudFront provides additional layers of security with DDoS protection, SSL/TLS encryption, and integration with AWS Web Application Firewall (WAF), ensuring the website is secure and resilient against common web threats.

- **Caching and Performance:** With CloudFront's caching mechanisms, frequently accessed content is stored at edge locations, reducing load on the S3 bucket and speeding up delivery to users.

Link to customer website: `d32uficm9gfop7.cloudfront.net`

Link to admin portal: `dfrtqijo7ap3x.cloudfront.net`

6.1 Optimizations for Performance and Efficiency

6.1.1 Origin Shield

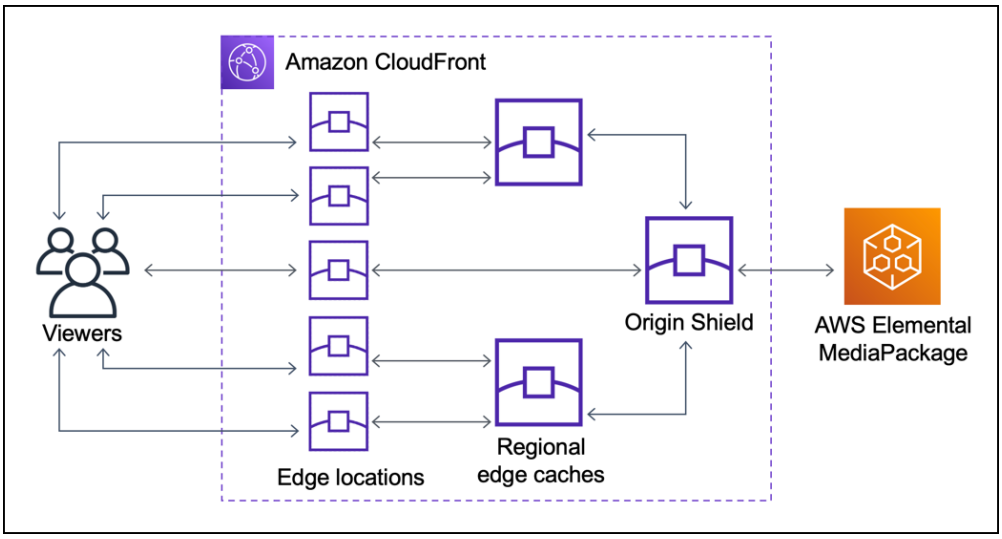


Figure 6.1: Functionality of Origin Shield

Origin Shield was activated in Amazon CloudFront to enhance caching efficiency and reduce the load on the origin server, which in this case is the S3 bucket hosting the website. Origin Shield serves as an additional layer between CloudFront and the origin, providing the following key benefits:

- **Reduced Origin Load:** By acting as a centralized caching layer, Origin Shield minimizes the number of requests that go directly to the S3 bucket. This helps to reduce costs related to origin requests and improve overall system performance.
- **Increased Cache Hit Ratio:** Origin Shield improves the cache hit ratio by consolidating and managing requests from multiple edge locations. This means more users will receive cached content without hitting the origin server, improving response times.
- **Cost Optimization:** Fewer requests reaching the origin result in reduced data transfer costs and operational load, offering a cost-effective optimization for handling web traffic.

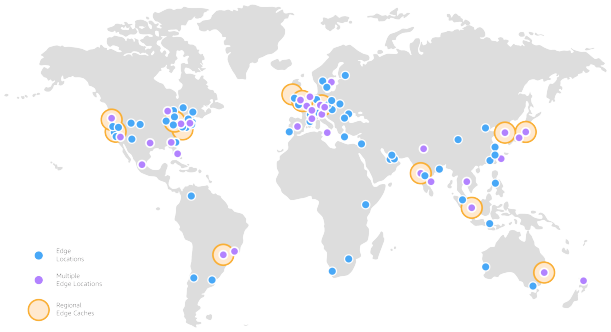


Figure 6.3: CloudFront edge locations

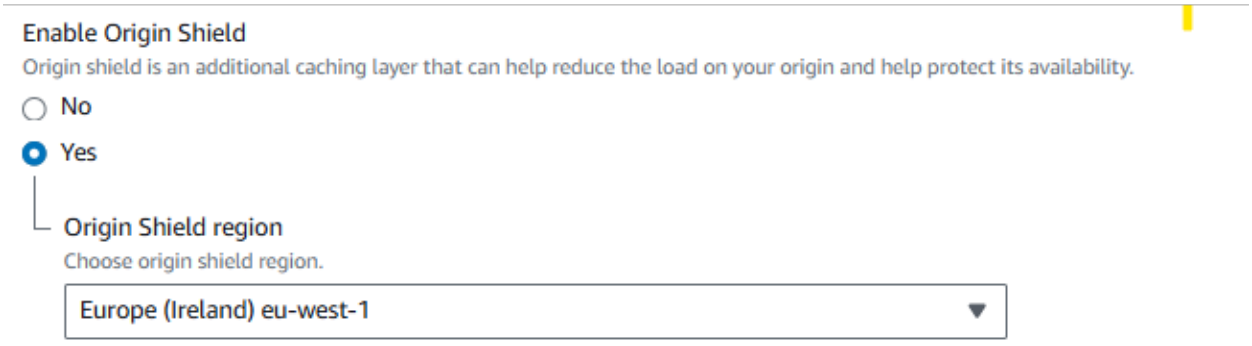


Figure 6.2: Origin Shield settings

6.1.2 Edge Locations

CloudFront was configured to use All Edge Locations for the best possible performance. This setting ensures that content is delivered from the edge location closest to each user, offering the following advantages:

- **Reduced Latency:** By leveraging all available edge locations, users receive content from the nearest geographical point, minimizing latency and improving website load times.

- **Enhanced User Experience:** Faster content delivery translates to a smoother and more responsive user experience, particularly important for a global audience where users may access the site from various regions.

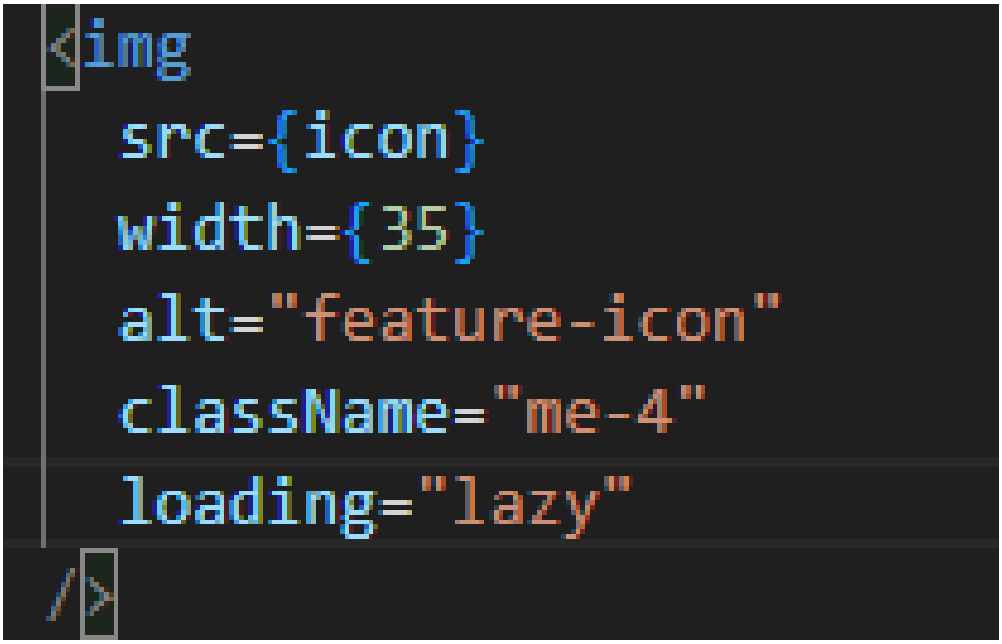


Figure 6.5: Lazy loading implementation

- **Scalability:** CloudFront's global network of edge locations ensures that the website can scale effectively, handling high traffic volumes while maintaining optimal performance, regardless of user location.

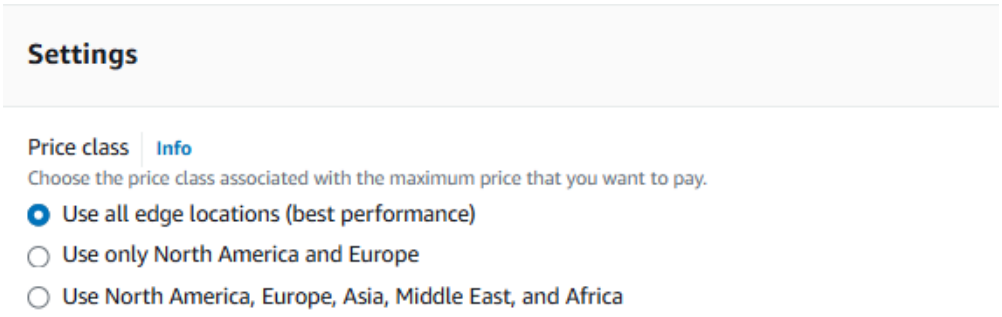


Figure 6.4: Origin Shield settings

6.1.3 Lazy Loading

Lazy loading was implemented to delay the loading of images in the web application. This was done to improve performance by reducing the initial load time.

Chapter 7

Conclusion

This report outlines the design, development, and deployment of a highly scalable and secure web application using AWS services. By leveraging a serverless architecture with AWS Lambda, API Gateway, and S3 the solution minimizes infrastructure management while ensuring optimal performance. The use of AWS CloudFront provides global distribution of the frontend with low-latency access, secured by AWS WAF and SSL encryption. Cognito ensures authentication for both customers and admins while IAM roles ensure secure access to other resources like RDS and S3.

DynamoDB and RDS enable efficient storage and retrieval of data while CloudWatch facilitates comprehensive monitoring and logging across the entire architecture.

Through careful consideration of security best practices, performance optimizations, and user management, this solution demonstrates how AWS services can be effectively combined to build a reliable, scalable, and secure platform for both customer and admin use cases.

Appendix A

Solution Design Diagram

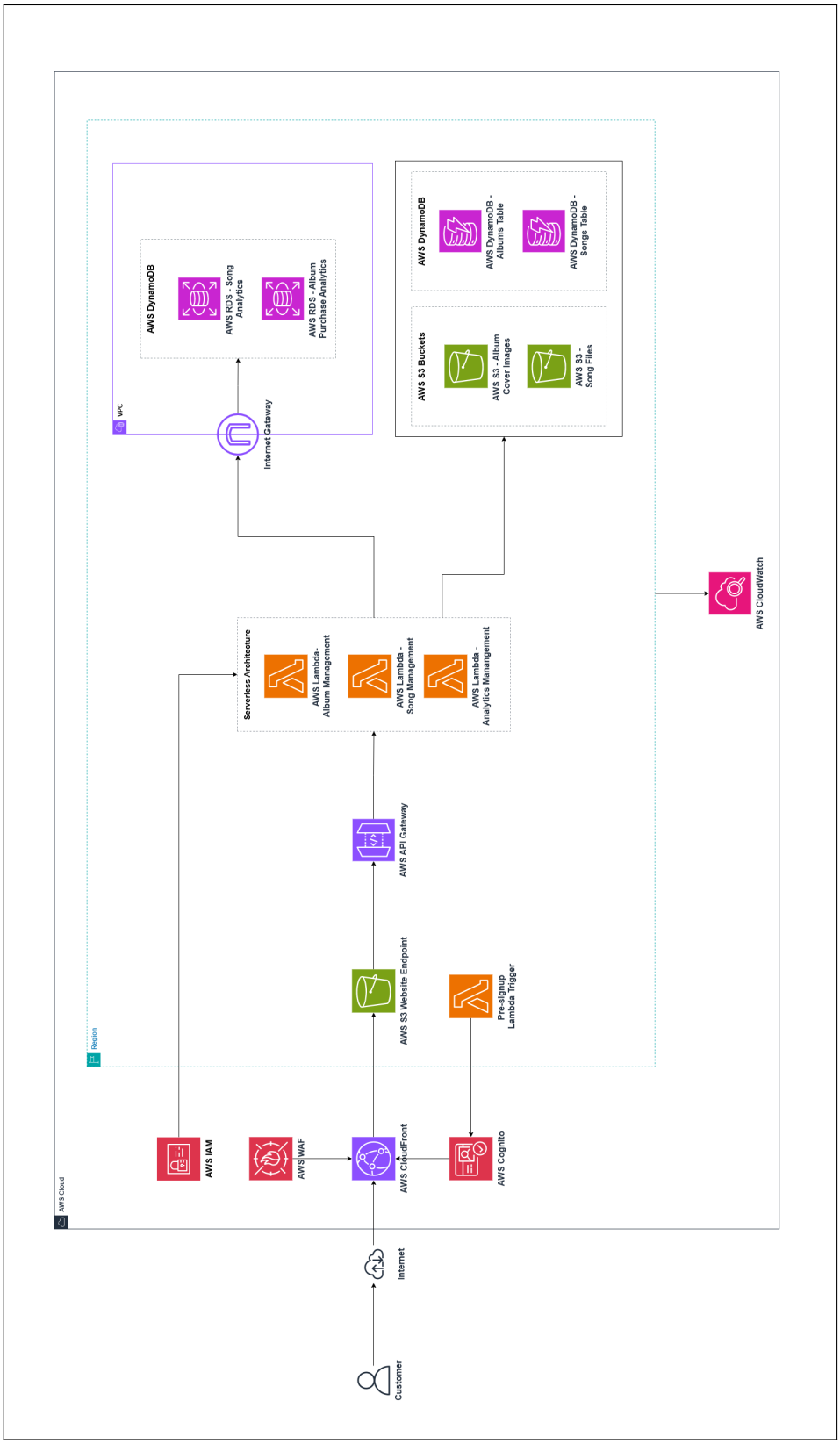


Figure A.1: AWS Solution Design Diagram for Customer Facing Website

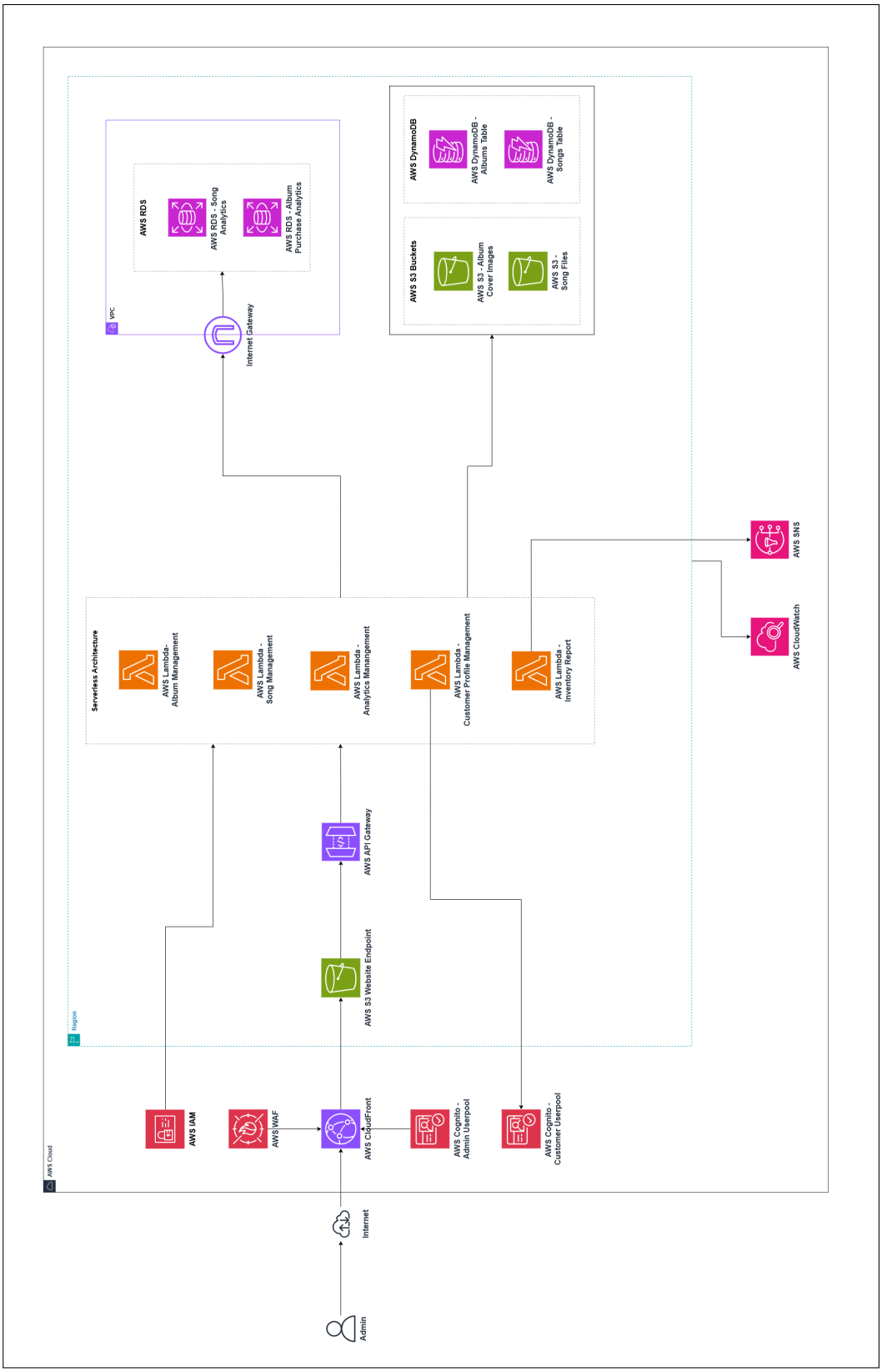


Figure A.2: AWS Solution Design Diagram for Admin Portal

Appendix B

Frontend UI Images

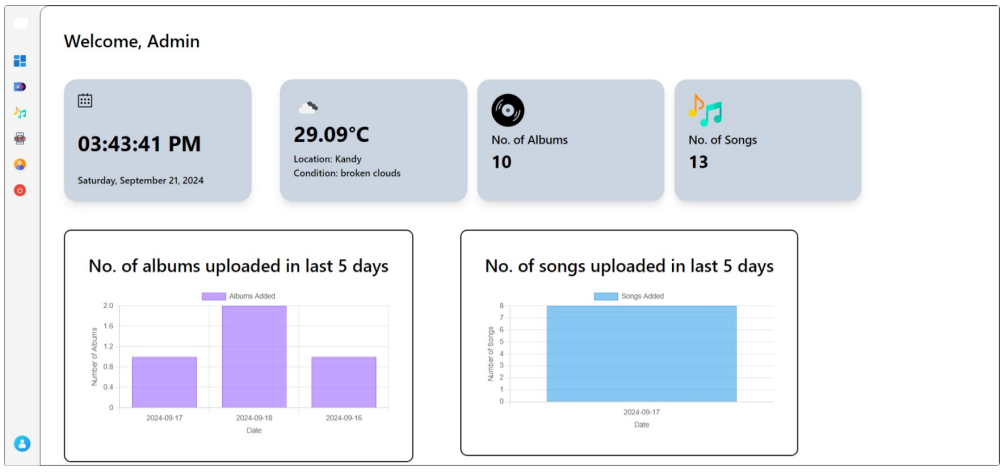


Figure B.12: Admin Home Page

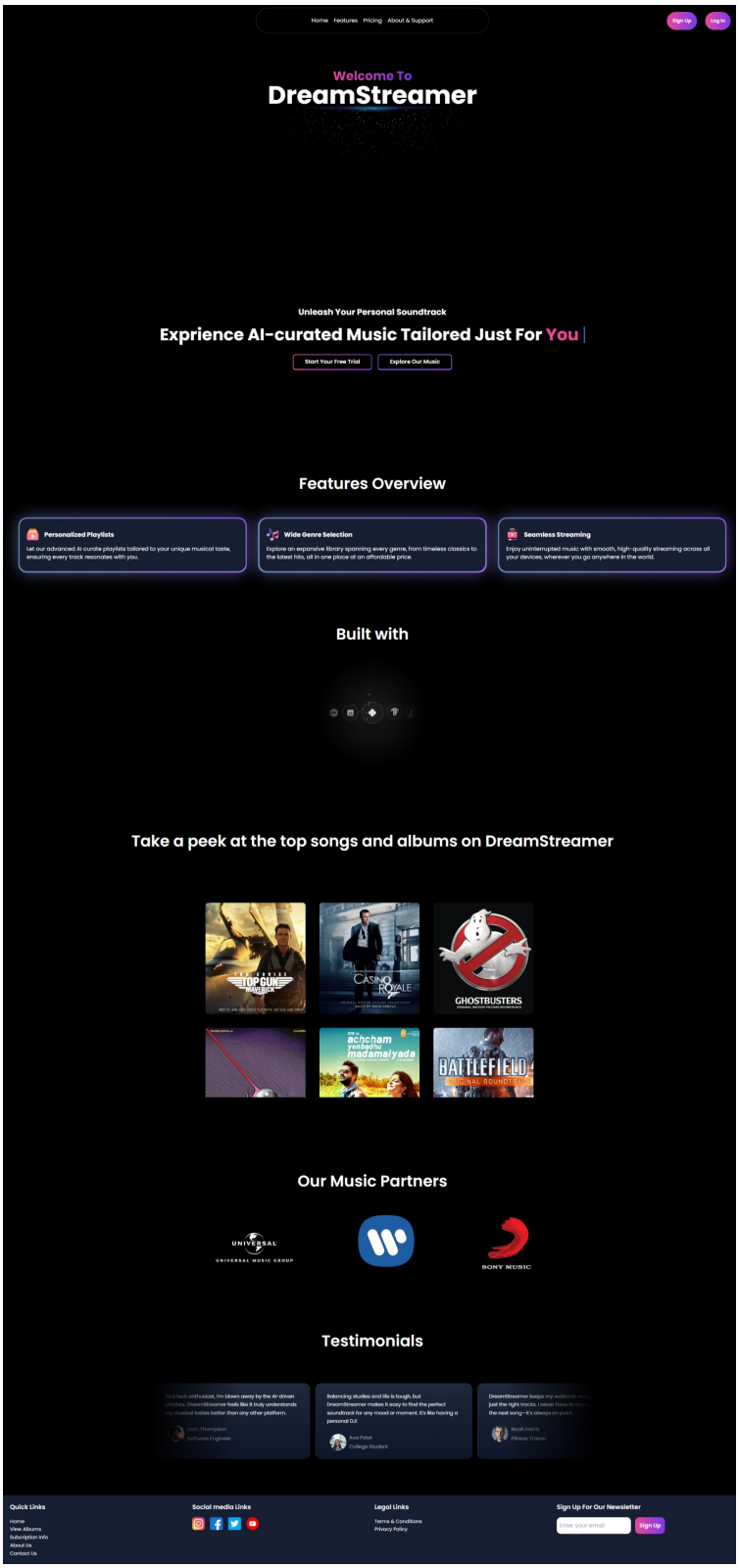


Figure B.1: Home Page

Figure B.2: Features Page

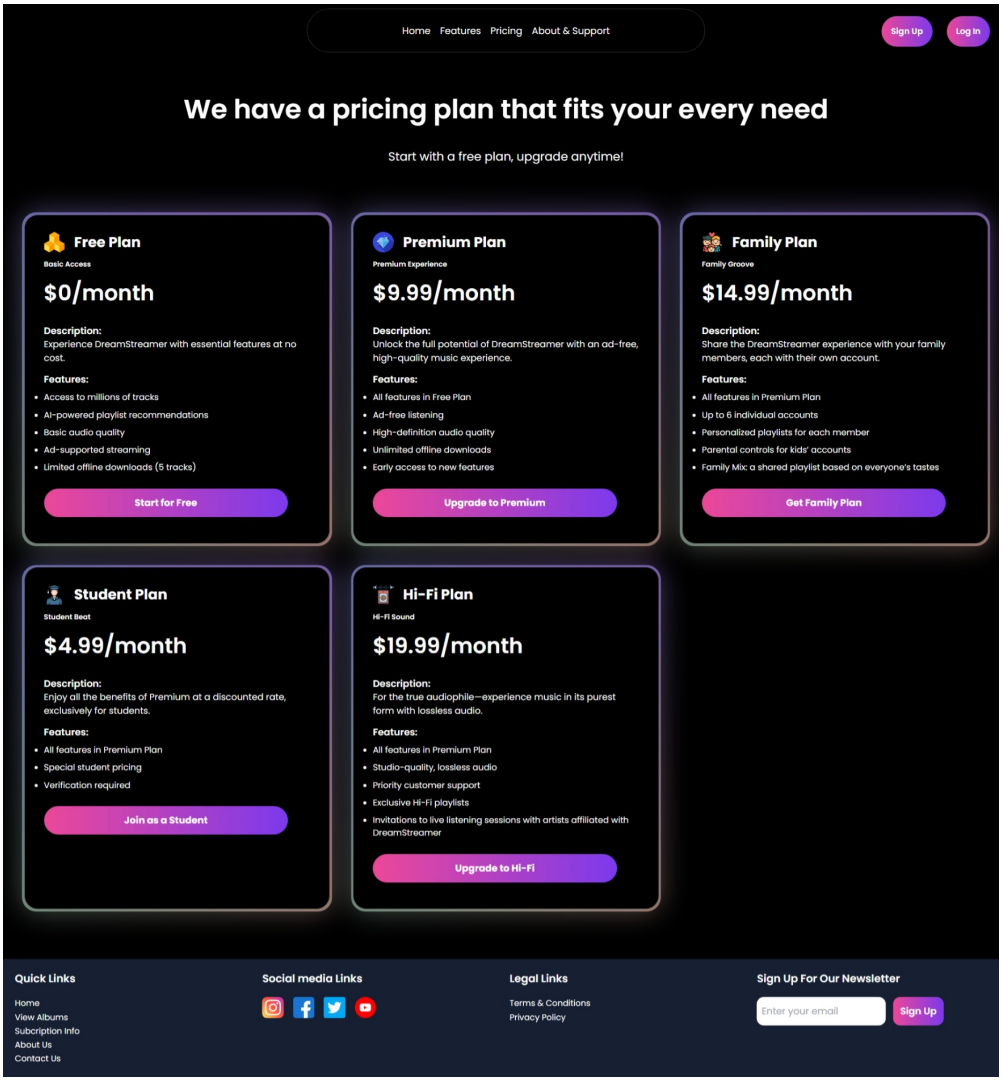


Figure B.3: Pricing Page

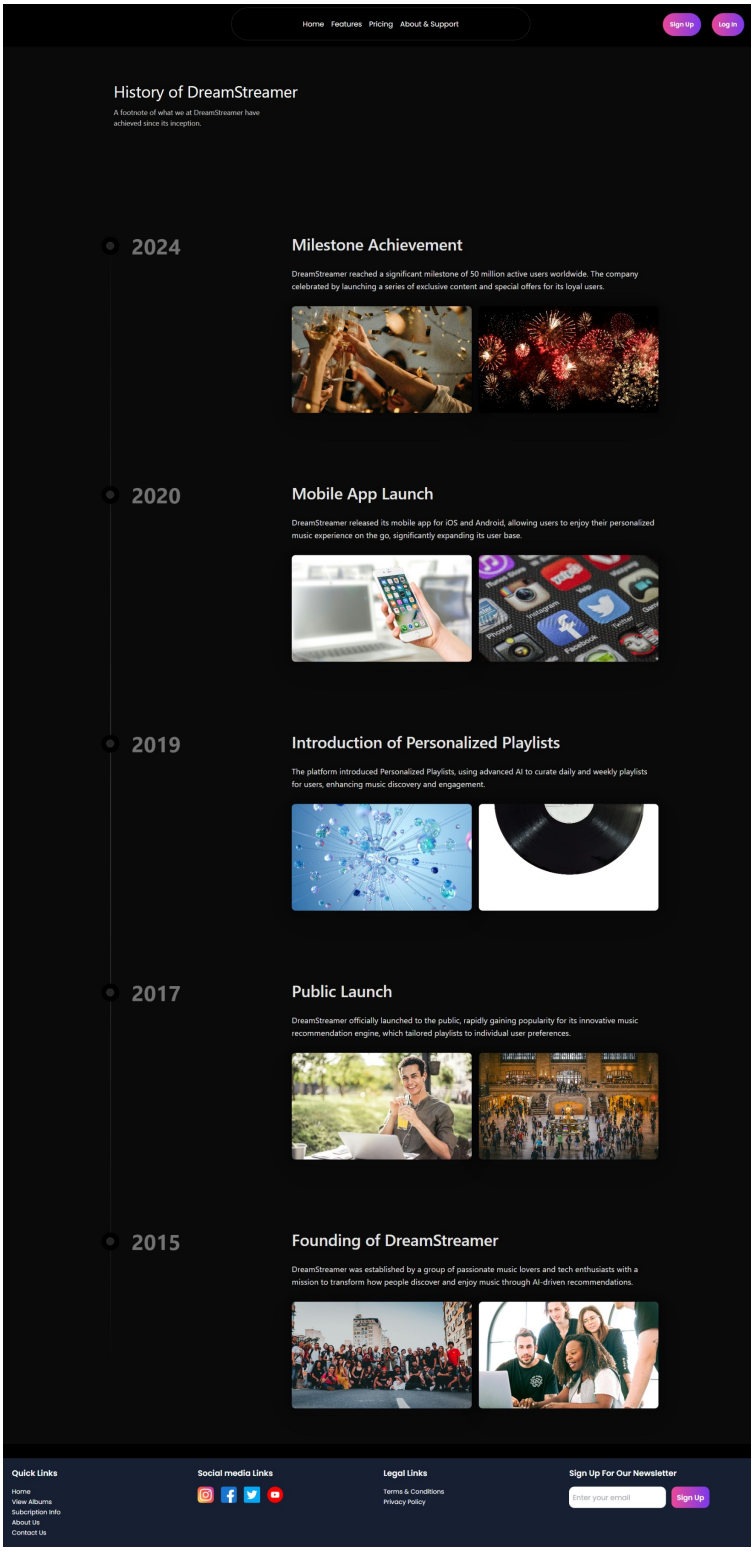


Figure B.4: About Page

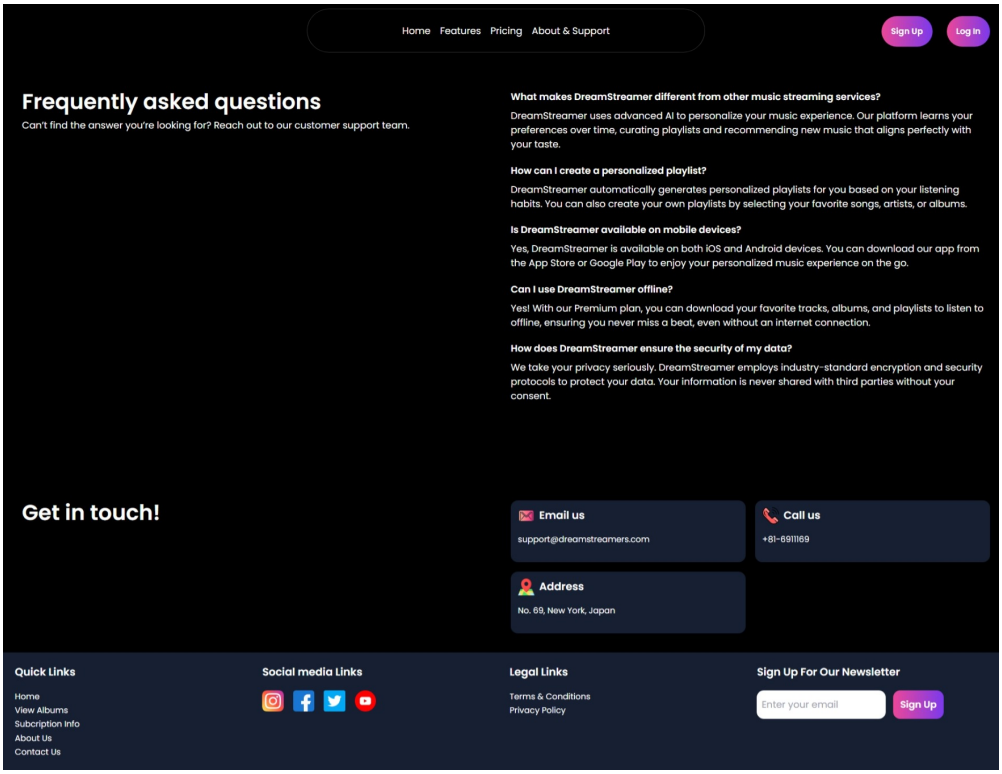


Figure B.5: Support Page

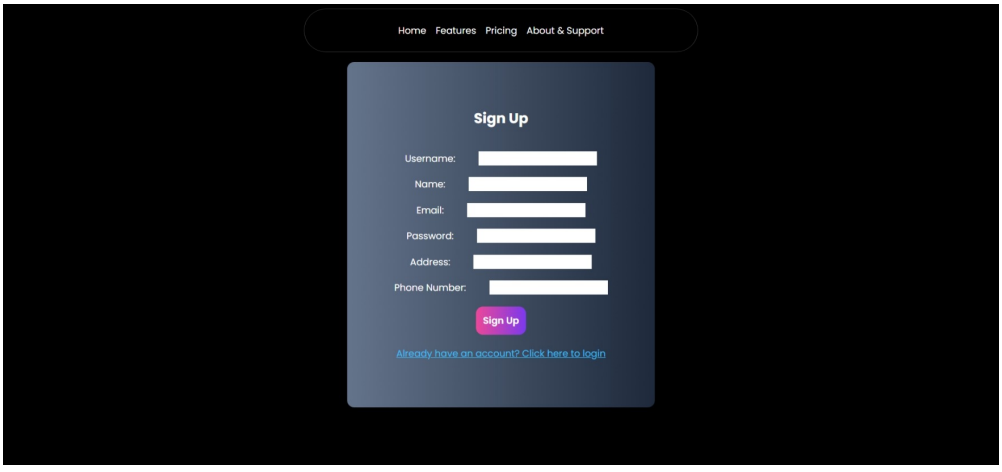


Figure B.6: Signup Page

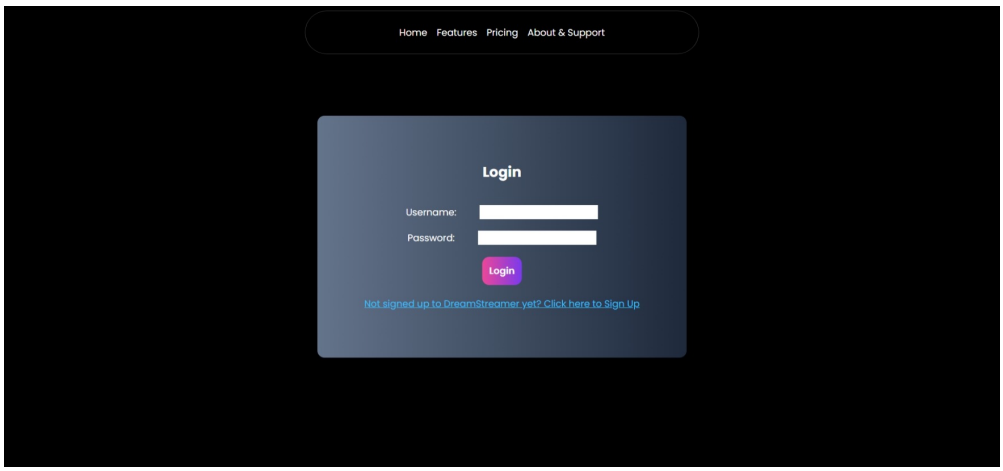


Figure B.7: Login Page

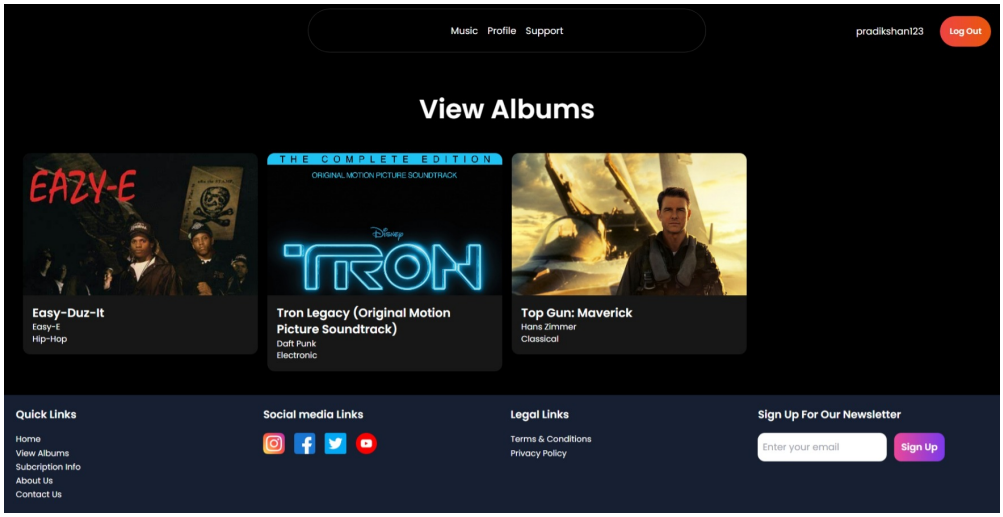


Figure B.8: Albums Page

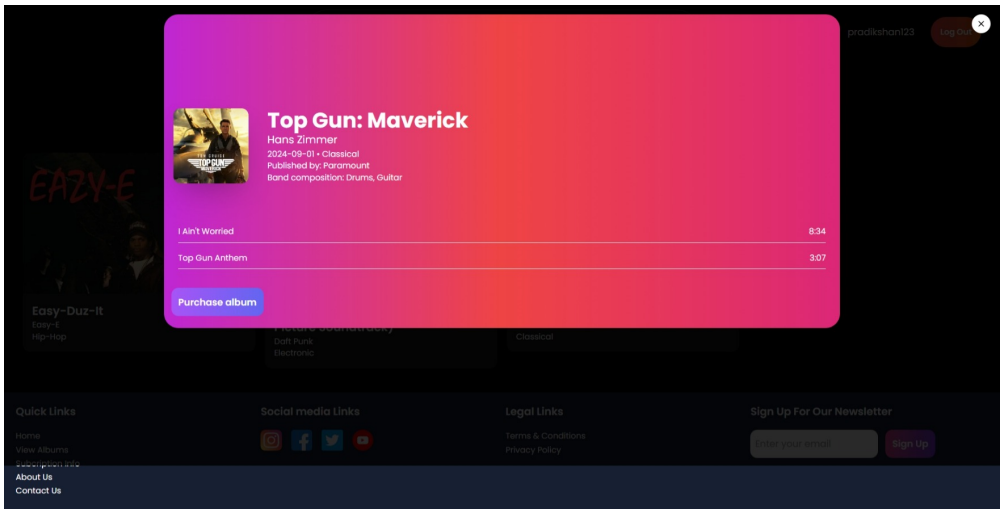


Figure B.9: Albums Detail Component

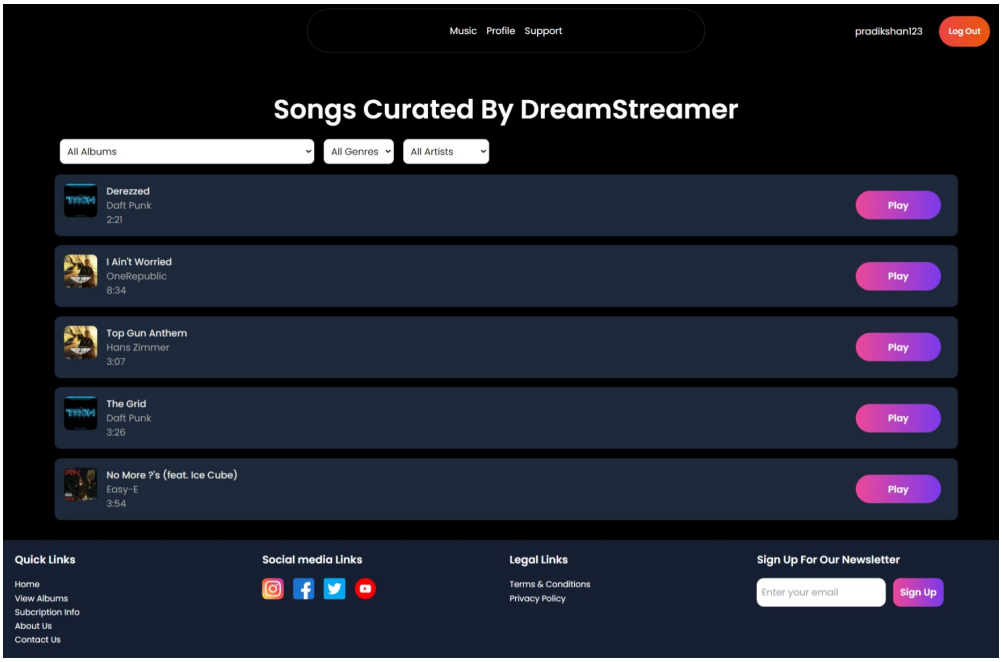


Figure B.10: Songs Page

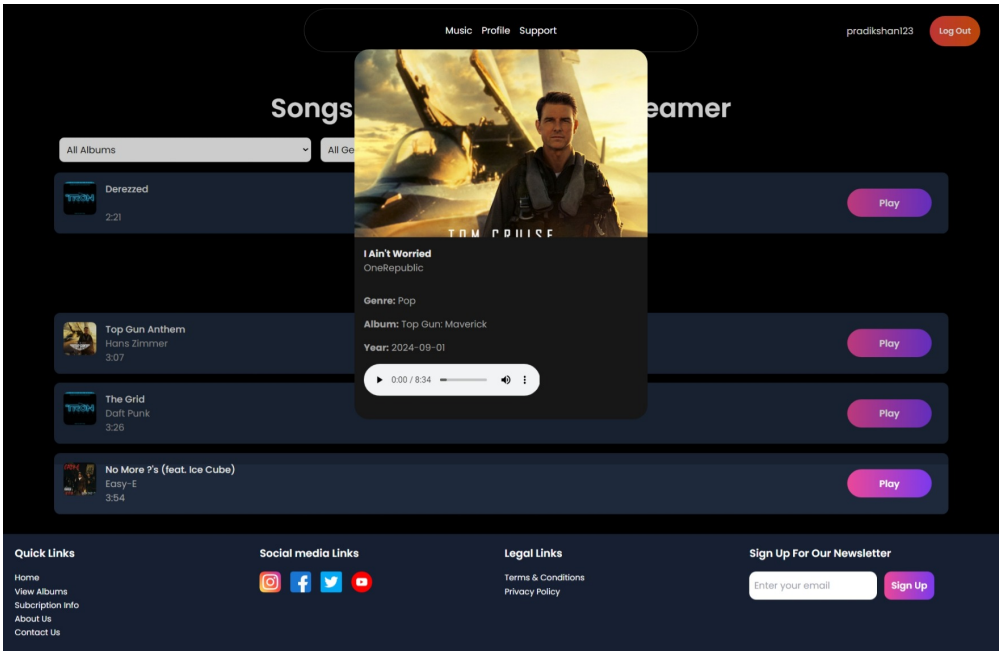


Figure B.11: Songs Detail Component

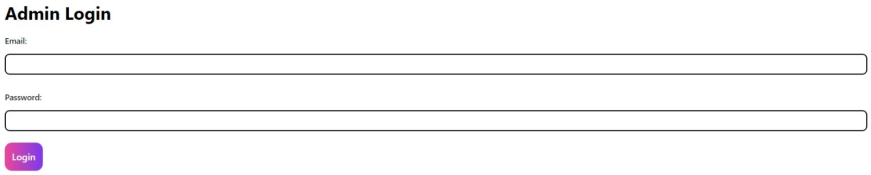


Figure B.13: Admin Login Page

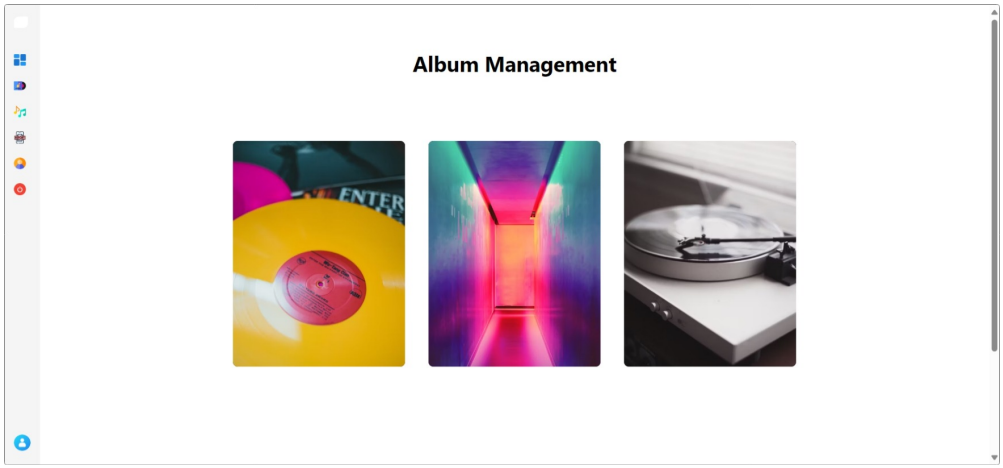


Figure B.14: Admin Album Management Page

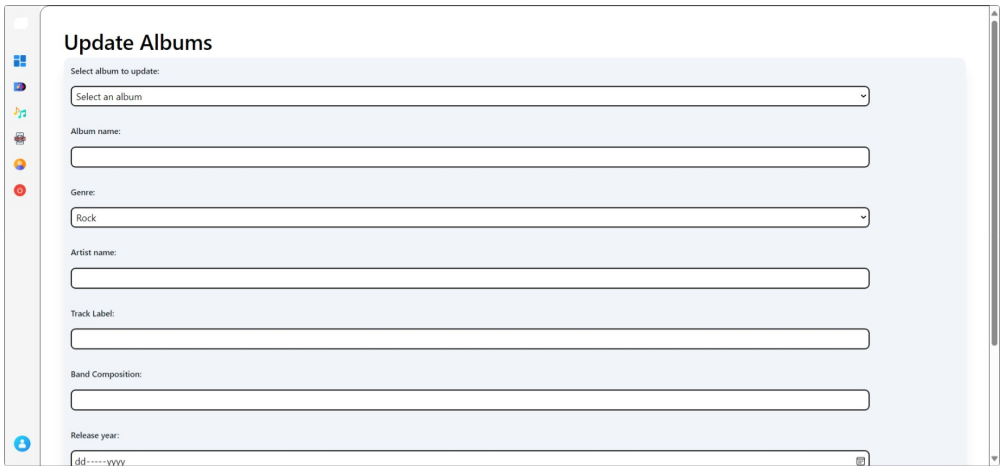


Figure B.16: Admin Update Album Page

Figure B.15: Admin Add Album Page

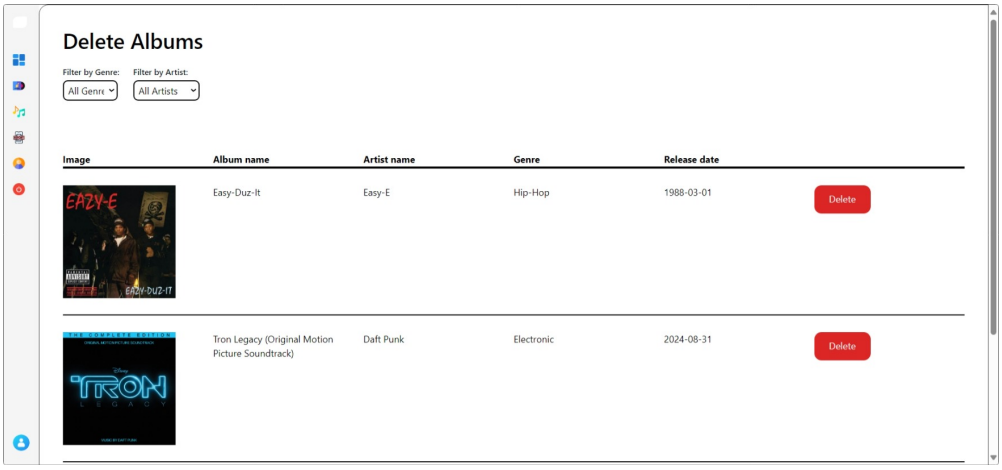


Figure B.17: Admin Delete Album Page

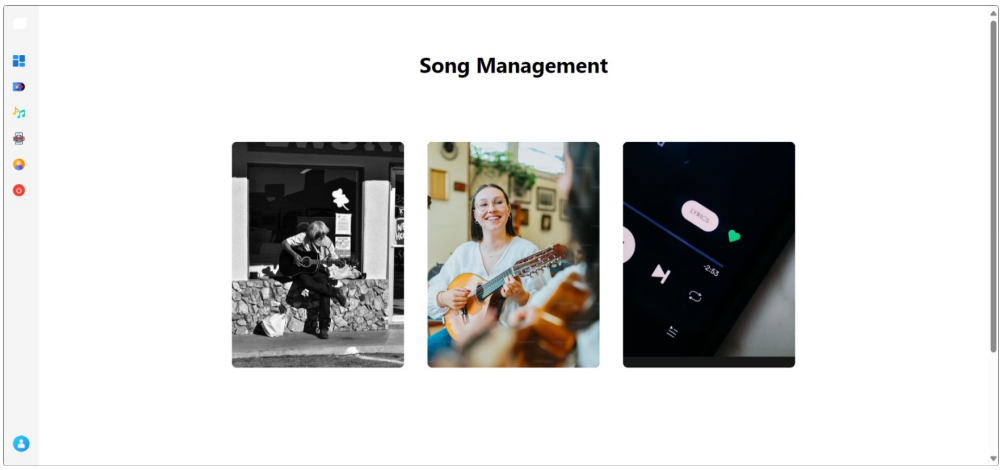


Figure B.18: Admin Song Management Page

Figure B.19: Admin Add Song Page

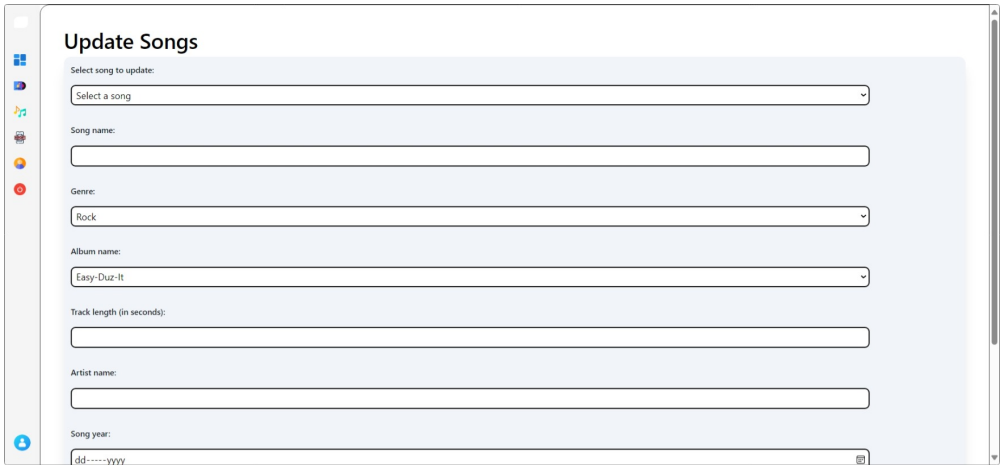


Figure B.20: Admin Update Song Page

[H]

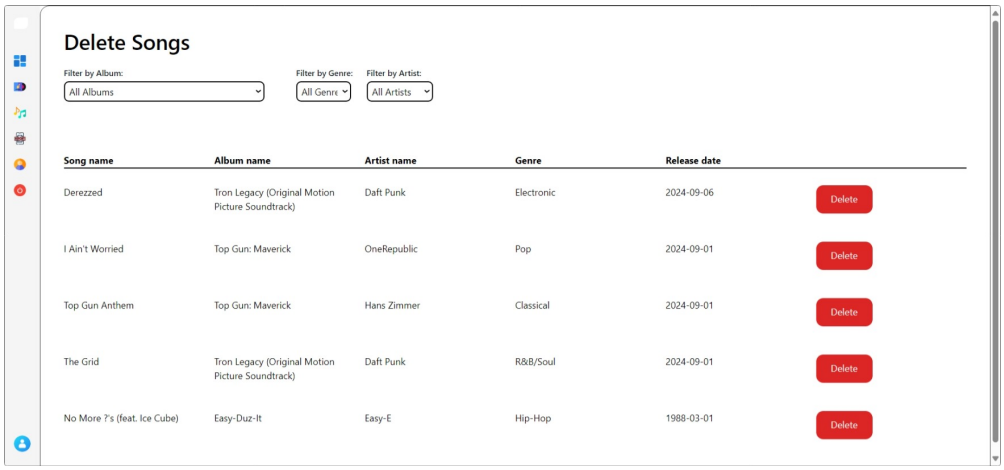


Figure B.21: Admin Delete Song Page

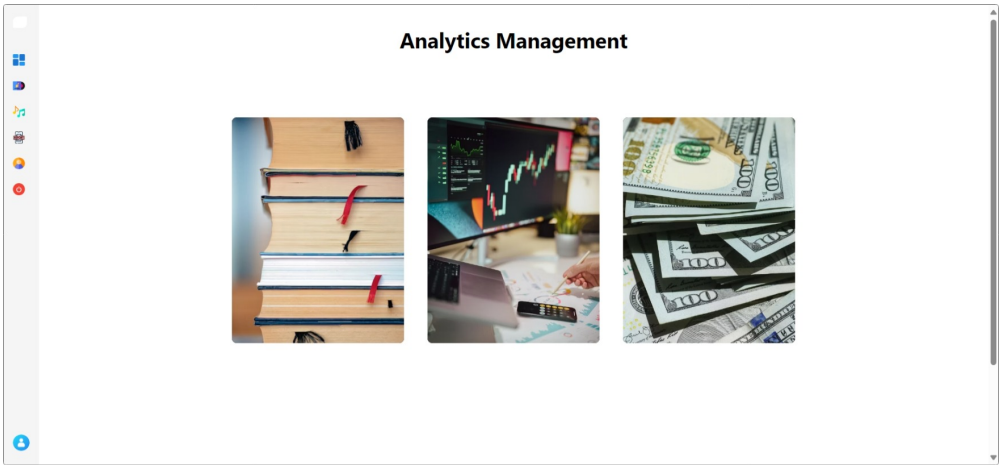


Figure B.22: Admin Analytics Management Page

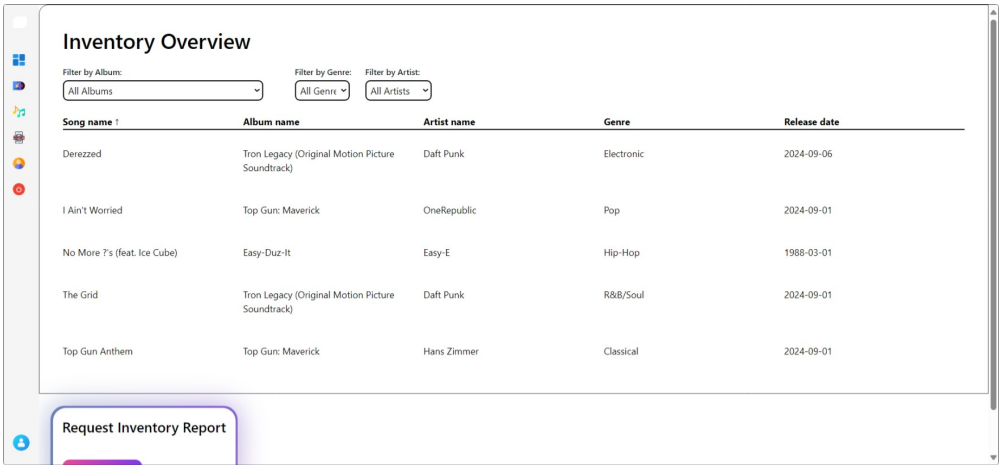


Figure B.23: Admin Inventory Overview Page

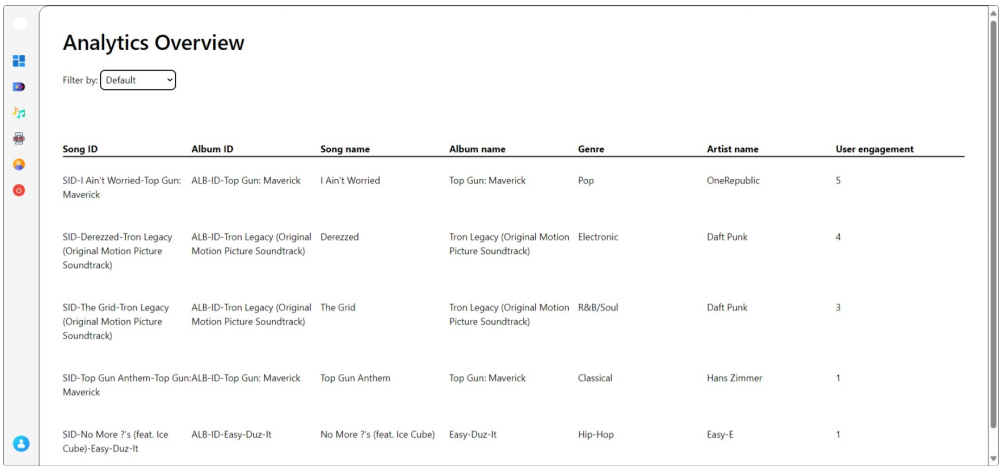


Figure B.24: Admin Analytics Overview Page

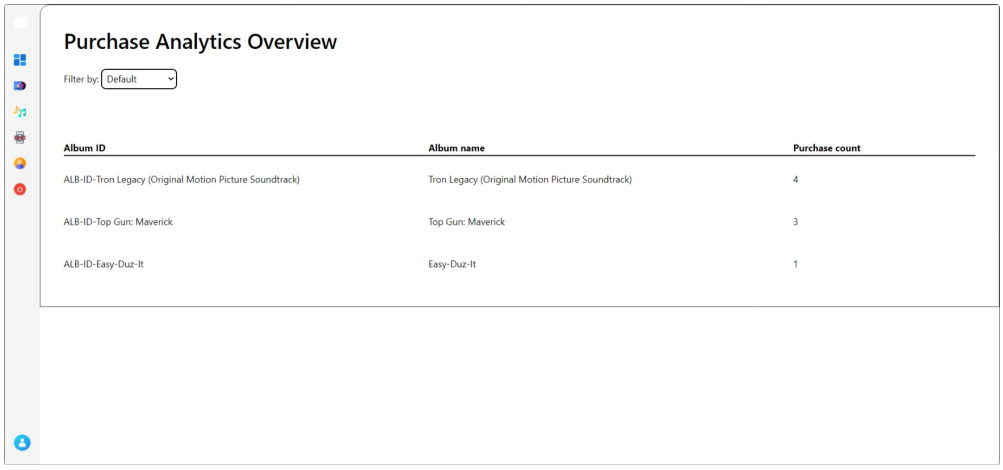


Figure B.25: Admin Purchase Analytics Overview Page

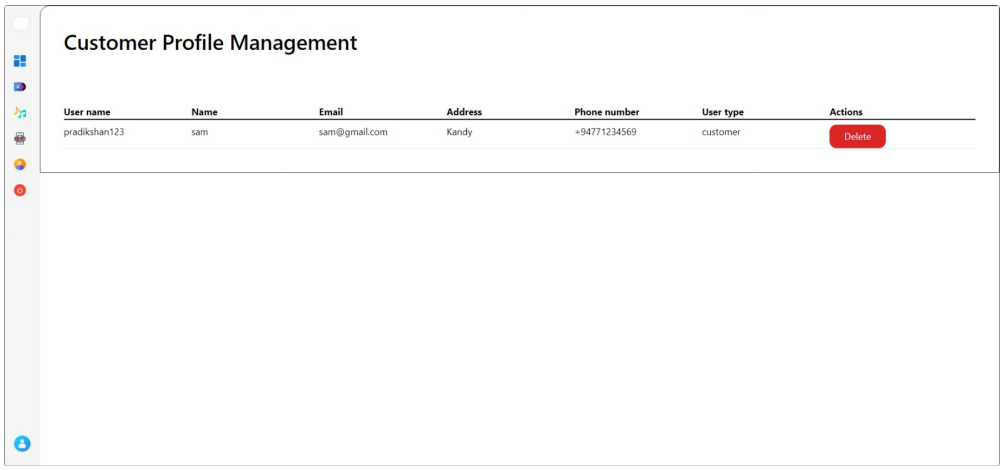


Figure B.26: Admin Customer Profile Management Page

Appendix C

Lambda Functions

```
import AWS from "aws-sdk";

const dynamoDB = new AWS.DynamoDB.DocumentClient();
const s3 = new AWS.S3();

export const handler = async (event) => {
  if (event.httpMethod === "OPTIONS") {
    return {
      statusCode: 200,
      body: JSON.stringify({ message: "CORS check passed" }),
    };
  }

  const {
    albumName,
    genre,
    artistName,
    trackLabel,
    bandComposition,
    albumYear,
    fileName,
    fileType,
  } = JSON.parse(event.body);

  if (!fileName || !fileType) {
    console.error("fileName or fileType is missing.");
    return {
      statusCode: 400,
      body: JSON.stringify({ error: "File name and type are required." }),
    };
  }

  if (
    !albumName ||
    !genre ||
    !artistName ||
    !trackLabel ||
    !bandComposition ||
    !albumYear
  ) {
    console.error("Missing fields");
    return {
      statusCode: 400,
      body: JSON.stringify({ error: "File name and type are required." }),
    };
  }
```

Figure C.1: addAlbum lambda function - 1

```
const albumId = `ALB-ID-${albumName}`;

const s3Params = {
  Bucket: "dreamstreamer-album-art",
  Key: fileName,
  Expires: 60 * 5,
  ContentType: fileType,
};

let uploadUrl = "";

try {
  uploadUrl = s3.getSignedUrl("putObject", s3Params);
} catch (error) {
  console.error("Error generating presigned URL:", error);
  return {
    statusCode: 500,
    body: JSON.stringify({ error: "Could not generate presigned URL" }),
  };
}

const permanentUrl = `https://dreamstreamer-album-art.s3.eu-west-1.amazonaws.com/${fileName}`;

const createdAt = new Date().toISOString();

const params = {
  TableName: "Albums",
  Item: {
    albumId: albumId,
    albumName: albumName,
    genre: genre,
    artistName: artistName,
    trackLabel: trackLabel,
    bandComposition: bandComposition,
    albumYear: albumYear,
    albumArtUrl: permanentUrl,
    createdAt: createdAt,
    updatedAt: createdAt,
  },
};
```

Figure C.2: addAlbum lambda function - 2

```
  try {
    await dynamoDB.put(params).promise();

    return {
      statusCode: 200,
      body: JSON.stringify({
        message: "Album added successfully!",
        presignedUrl: uploadUrl,
      }),
    };
  } catch (error) {
    console.error("Error adding album:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not add album" }),
    };
  }
};
```

Figure C.3: addAlbum lambda function - 3

```
import AWS from "aws-sdk";

const dynamoDB = new AWS.DynamoDB.DocumentClient();

export const handler = async (event) => {
  const params = {
    TableName: "Albums",
  };

  try {
    const data = await dynamoDB.scan(params).promise();

    return {
      statusCode: 200,
      body: JSON.stringify(data.Items),
    };
  } catch (error) {
    console.error("Error fetching album names:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not fetch album names" }),
    };
  }
};
```

Figure C.4: getAlbumDetails lambda function

```
import AWS from "aws-sdk";

const dynamoDB = new AWS.DynamoDB.DocumentClient();

export const handler = async (event) => {
  const albumId = event.queryStringParameters.albumId;

  const params = {
    TableName: "Albums",
    Key: {
      albumId: albumId,
    },
  };

  try {
    const data = await dynamoDB.get(params).promise();
    if (!data.Item) {
      return {
        statusCode: 404,
        body: JSON.stringify({ error: "Album not found" }),
      };
    }
    return {
      statusCode: 200,
      body: JSON.stringify({ albumArtUrl: data.Item.albumArtUrl }),
    };
  } catch (error) {
    console.error("Error fetching album cover:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not fetch album cover" }),
    };
  }
};
```

Figure C.5: getAlbumCover lambda function

```
import AWS from "aws-sdk";

const dynamoDb = new AWS.DynamoDB.DocumentClient();

export const handler = async (event) => {
  const albumId = event.queryStringParameters.albumId;

  const params = {
    TableName: "Albums",
    Key: {
      albumId: albumId,
    },
  };

  try {
    const data = await dynamoDb.get(params).promise();
    if (data.Item) {
      return {
        statusCode: 200,
        body: JSON.stringify(data.Item),
      };
    } else {
      return {
        statusCode: 404,
        body: JSON.stringify({ error: "Album not found" }),
      };
    }
  } catch (error) {
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not retrieve album" }),
    };
  }
};
```

Figure C.6: getAlbumDetailsById lambda function

```
import AWS from "aws-sdk";
const dynamoDb = new AWS.DynamoDB.DocumentClient();

export const handler = async (event) => {
  const albumId = event.queryStringParameters.albumId;
  const requestBody = JSON.parse(event.body);

  const updatedAt = new Date().toISOString();

  let updateExpression =
    "set albumName = :albumName, genre = :genre,  artistName = :artistName, trackLabel = :trackLabel, bandComposition = :bandComposition, albumYear = :albumYear, updatedAt =
    :updatedAt";
  let expressionAttributeValues = {
    ":albumName": requestBody.albumName,
    ":genre": requestBody.genre,
    ":artistName": requestBody.artistName,
    ":trackLabel": requestBody.trackLabel,
    ":bandComposition": requestBody.bandComposition,
    ":albumYear": requestBody.albumYear,
    ":updatedAt": updatedAt,
  };

  if (requestBody.albumArtUrl) {
    updateExpression += ", albumArtUrl = :albumArtUrl";
    expressionAttributeValues[":albumArtUrl"] = requestBody.albumArtUrl;
  }

  const params = {
    TableName: "Albums",
    Key: { albumId: albumId },
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW",
  };

  try {
    const result = await dynamoDb.update(params).promise();
    return {
      statusCode: 200,
      body: JSON.stringify(result.Attributes),
    };
  } catch (error) {
    console.error(error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not update album" }),
    };
  }
}
```

Figure C.7: updateAlbum lambda function

```
import AWS from "aws-sdk";
const s3 = new AWS.S3();

export const handler = async (event) => {
  const { fileName, fileType } = JSON.parse(event.body);
  const s3Params = {
    Bucket: "dreamstreamer-album-art",
    Key: `${fileName}`,
    Expires: 60 * 5,
    ContentType: fileType,
    // ACL: "public-read",
  };

  try {
    const presignedUrl = await s3.getSignedUrlPromise("putObject", s3Params);
    console.log("File to upload: ", s3Params);
    return {
      statusCode: 200,
      body: JSON.stringify({
        presignedUrl: presignedUrl,
        fileUrl: `https://dreamstreamer-album-art.s3.eu-west-1.amazonaws.com/${fileName}`,
      }),
    };
  } catch (error) {
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not generate presigned URL" }),
    };
  }
};
```

Figure C.8: uploadNewAlbumCover lambda function

```
import AWS from "aws-sdk";
const dynamoDb = new AWS.DynamoDB.DocumentClient();
const s3 = new AWS.S3();

export const handler = async (event) => {
  const { albumId, albumArtUrl } = JSON.parse(event.body);

  const albumParams = {
    TableName: "Albums",
    Key: {
      albumId: albumId,
    },
  };

  const querySongsParams = {
    TableName: "Songs",
    IndexName: "albumId-songId-index",
    KeyConditionExpression: "albumId = :albumId",
    ExpressionAttributeValues: {
      ":albumId": albumId,
    },
  };

  try {
    await dynamoDb.delete(albumParams).promise();

    const songsData = await dynamoDb.query(querySongsParams).promise();

    if (songsData.Items && songsData.Items.length > 0) {
      const deleteSongPromises = songsData.Items.map((song) => {
        const deleteSongParams = {
          TableName: "Songs",
          Key: {
            songId: song.songId,
          },
        };
        return dynamoDb.delete(deleteSongParams).promise();
      });

      await Promise.all(deleteSongPromises);
    }

    const fileName = albumArtUrl.split("/").pop();
    const s3Params = {
      Bucket: "dreamstreamer-album-art",
      Key: fileName,
    };
```

Figure C.9: deleteAlbum lambda function

```
import AWS from "aws-sdk";

const dynamoDB = new AWS.DynamoDB.DocumentClient();
const s3 = new AWS.S3();

export const handler = async (event) => {
  const {
    songName,
    genre,
    albumName,
    trackLength,
    artistName,
    songYear,
    fileName,
    fileType,
  } = JSON.parse(event.body);

  console.log("Received albumName:", albumName);

  if (!fileName || !fileType) {
    console.error("fileName or fileType is missing.");
    return {
      statusCode: 400,
      body: JSON.stringify({ error: "File name and type are required." }),
    };
  }

  const songId = `SID-${songName}-${albumName}`;

  console.log("Querying album with name:", albumName);

  let albumId;
  const albumParams = {
    TableName: "Albums",
    IndexName: "albumName-index",
    KeyConditionExpression: "albumName = :albumName",
    ExpressionAttributeValues: {
      ":albumName": albumName,
    },
  };
```

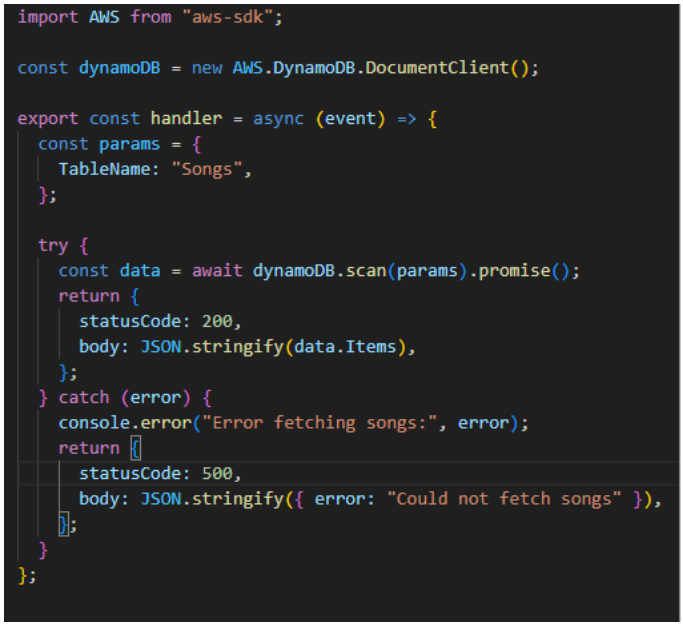
Figure C.10: addSong lambda function - 1

```
try {
  const albumResult = await dynamoDB.query(albumParams).promise();
  albumId =
    albumResult.Items && albumResult.Items.length > 0
      ? albumResult.Items[0].albumId
      : null;

  if (!albumId) {
    return {
      statusCode: 400,
      body: JSON.stringify({ error: "Album not found." }),
    };
  }
} catch (error) {
  console.error("Error fetching album:", error);
  return {
    statusCode: 500,
    body: JSON.stringify({ error: "Could not fetch album details." }),
  };
}

const s3Params = {
  Bucket: "dreamstreamer-song",
  Key: fileName,
  Expires: 60 * 5,
  ContentType: fileType,
};

let songFileUrl = "";

try {
  songFileUrl = s3.getSignedUrl("putObject", s3Params);
  console.log("Generated presigned URL:", songFileUrl);
} catch (error) {
  console.error("Error generating presigned URL:", error);
  return {
    statusCode: 500,
    body: JSON.stringify({ error: "Could not generate presigned URL" }),
  };
}
```

Figure C.11: addSong lambda function - 2

```
  const params = {
    TableName: "Songs",
    Item: {
      songId: songId,
      songName: songName,
      genre: genre,
      albumName: albumName,
      albumId: albumId,
      trackLength: trackLength,
      artistName: artistName,
      songYear: songYear,
      songFileUrl: songFileUrl.split("?")[0],
      createdAt: createdAt,
      updatedAt: createdAt,
    },
  };

  try {
    await dynamoDB.put(params).promise();

    return {
      statusCode: 200,
      body: JSON.stringify({
        message: "Song added successfully!",
        presignedUrl: songFileUrl,
      }),
    };
  } catch (error) {
    console.error("Error adding song:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not add song" }),
    };
  }
};
```

Figure C.12: addSong lambda function - 3



```
import AWS from "aws-sdk";

const dynamoDB = new AWS.DynamoDB.DocumentClient();

export const handler = async (event) => {
  const params = {
    TableName: "Songs",
  };

  try {
    const data = await dynamoDB.scan(params).promise();
    return {
      statusCode: 200,
      body: JSON.stringify(data.Items),
    };
  } catch (error) {
    console.error("Error fetching songs:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not fetch songs" }),
    };
  }
};
```

Figure C.13: getSongDetails lambda function

```
import AWS from "aws-sdk";

const dynamoDB = new AWS.DynamoDB.DocumentClient();

export const handler = async (event) => {
  const albumName = event.queryStringParameters?.albumName;

  if (!albumName) {
    return {
      statusCode: 400,
      body: JSON.stringify({ error: "Missing albumName parameter" }),
    };
  }

  const params = {
    TableName: "Songs",
    IndexName: "albumName-songId-index",
    KeyConditionExpression: "albumName = :albumName",
    ExpressionAttributeValues: {
      ":albumName": albumName,
    },
  };

  try {
    const data = await dynamoDB.query(params).promise();
    console.log("Data:", JSON.stringify(data));
    return {
      statusCode: 200,
      body: JSON.stringify(data.Items || []),
    };
  } catch (error) {
    console.error("Error fetching songs:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not fetch songs" }),
    };
  }
};
```

Figure C.14: getSongForAlbum lambda function

```
import AWS from "aws-sdk";

const dynamoDb = new AWS.DynamoDB.DocumentClient();

export const handler = async (event) => {
  const songId = event.queryStringParameters.songId;

  const params = {
    TableName: "Songs",
    Key: {
      songId: songId,
    },
  };

  try {
    const data = await dynamoDb.get(params).promise();
    if (data.Item) {
      return {
        statusCode: 200,
        body: JSON.stringify(data.Item),
      };
    } else {
      return {
        statusCode: 404,
        body: JSON.stringify({ error: "Song not found" }),
      };
    }
  } catch (error) {
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not retrieve song" }),
    };
  }
};
```

Figure C.15: getSongDetailsById lambda function

```
import AWS from "aws-sdk";
const dynamoDb = new AWS.DynamoDB.DocumentClient();

export const handler = async (event) => {
  const songId = event.queryStringParameters.songId;
  const requestBody = JSON.parse(event.body);

  const updatedAt = new Date().toISOString();

  let updateExpression =
    "set songName = :songName, genre = :genre, albumName = :albumName, trackLength = :trackLength, artistName = :artistName, songYear = :songYear, updatedAt = :updatedAt";
  let expressionAttributeValues = {
    ":songName": requestBody.songName,
    ":genre": requestBody.genre,
    ":albumName": requestBody.albumName,
    ":trackLength": requestBody.trackLength,
    ":artistName": requestBody.artistName,
    ":songYear": requestBody.songYear,
    ":updatedAt": updatedAt,
  };

  if (requestBody.songFileUrl) {
    updateExpression += ", songFileUrl = :songFileUrl";
    expressionAttributeValues[":songFileUrl"] = requestBody.songFileUrl;
  }

  const params = {
    TableName: "Songs",
    Key: { songId: songId },
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW",
  };

  try {
    const result = await dynamoDb.update(params).promise();
    return {
      statusCode: 200,
      body: JSON.stringify(result.Attributes),
    };
  } catch (error) {
    console.error(error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not update song" }),
    };
  }
};
```

Figure C.16: songUpdate lambda function

```
import AWS from "aws-sdk";
const s3 = new AWS.S3();

export const handler = async (event) => {
  const { fileName, fileType } = JSON.parse(event.body);
  const s3Params = {
    Bucket: "dreamstreamer-song",
    Key: `${fileName}`,
    Expires: 60 * 5,
    ContentType: fileType,
    // ACL: "public-read",
  };

  try {
    const presignedUrl = await s3.getSignedUrlPromise("putObject", s3Params);
    return {
      statusCode: 200,
      body: JSON.stringify({
        presignedUrl: presignedUrl,
        fileUrl: `https://dreamstreamer-song.s3.eu-west-1.amazonaws.com/${fileName}`,
      }),
    };
  } catch (error) {
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not generate presigned URL" }),
    };
  }
};
```

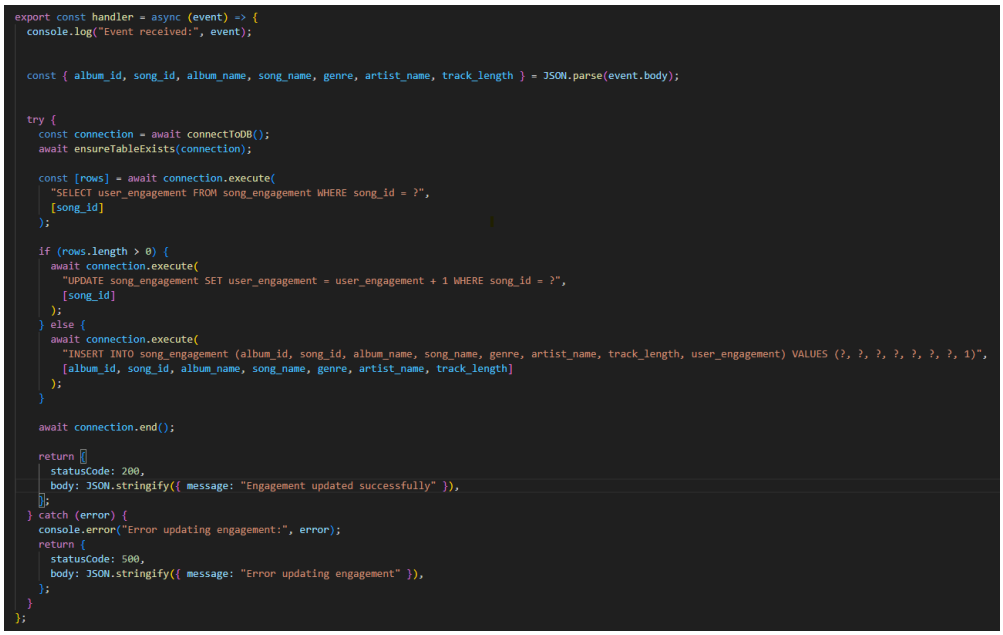
Figure C.17: uploadNewSong lambda function

```
import AWS from "aws-sdk";
const dynamoDb = new AWS.DynamoDB.DocumentClient();
const s3 = new AWS.S3();

export const handler = async (event) => {
  const { songId, songFileUrl } = JSON.parse(event.body);

  const params = {
    TableName: "Songs",
    Key: {
      songId: songId,
    },
  };

  try {
    await dynamoDb.delete(params).promise();

    const fileName = songFileUrl.split("/").pop();

    const s3Params = {
      Bucket: "dreamstreamer-song",
      Key: fileName,
    };

    await s3.deleteObject(s3Params).promise();

    return {
      statusCode: 200,
      body: JSON.stringify({ message: "Song deleted successfully" }),
    };
  } catch (error) {
    console.error(error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Failed to delete song" }),
    };
  }
};
```

Figure C.18: songDelete lambda function

```
import mysql from "mysql2/promise";

const DB_HOST = "cluster-1.czoccuoywyke.eu-west-1.rds.amazonaws.com";
const DB_USER = "admin";
const DB_PASSWORD = "RTX4070super";
const DB_NAME = "analytics";

const connectToDB = async () => {
  return await mysql.createConnection({
    host: DB_HOST,
    user: DB_USER,
    password: DB_PASSWORD,
    database: DB_NAME,
  });
};

const ensureTableExists = async (connection) => {
  const createTableQuery = `
    CREATE TABLE IF NOT EXISTS song_engagement (
      id INT AUTO_INCREMENT PRIMARY KEY,
      album_id VARCHAR(255),
      song_id VARCHAR(255) NOT NULL,
      album_name VARCHAR(255),
      song_name VARCHAR(255),
      genre VARCHAR(255),
      artist_name VARCHAR(255),
      track_length INT,
      user_engagement INT DEFAULT 0,
      UNIQUE(song_id)
    )
  `;

  try {
    await connection.execute(createTableQuery);
    console.log("Table 'song_engagement' ensured.");
  } catch (error) {
    console.error("Error ensuring table exists:", error);
    throw error;
  }
};
```

Figure C.19: addAnalytics lambda function - 1



```
export const handler = async (event) => {
  console.log("Event received:", event);


  const { album_id, song_id, album_name, song_name, genre, artist_name, track_length } = JSON.parse(event.body);


  try {
    const connection = await connectToDB();
    await ensureTableExists(connection);

    const [rows] = await connection.execute(
      "SELECT user_engagement FROM song_engagement WHERE song_id = ?",
      [song_id]
    );

    if (rows.length > 0) {
      await connection.execute(
        "UPDATE song_engagement SET user_engagement = user_engagement + 1 WHERE song_id = ?",
        [song_id]
      );
    } else {
      await connection.execute(
        "INSERT INTO song_engagement (album_id, song_id, album_name, song_name, genre, artist_name, track_length, user_engagement) VALUES (?, ?, ?, ?, ?, ?, ?, 1)",
        [album_id, song_id, album_name, song_name, genre, artist_name, track_length]
      );
    }

    await connection.end();

    return {
      statusCode: 200,
      body: JSON.stringify({ message: "Engagement updated successfully" }),
    };
  } catch (error) {
    console.error("Error updating engagement:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ message: "Error updating engagement" }),
    };
  }
};
```

Figure C.20: addAnalytics lambda function - 2

```
import mysql from 'mysql2/promise';

const DB_HOST = 'cluster-1.czoccuoywyke.eu-west-1.rds.amazonaws.com';
const DB_USER = 'admin';
const DB_PASSWORD = 'RTX4070super';
const DB_NAME = 'analytics';

const connectToDB = async () => {
  return await mysql.createConnection({
    host: DB_HOST,
    user: DB_USER,
    password: DB_PASSWORD,
    database: DB_NAME,
  });
};

const ensureTableExists = async (connection) => {
  const createTableQuery = `
    CREATE TABLE IF NOT EXISTS purchase_analytics (
      album_id VARCHAR(255) PRIMARY KEY,
      album_name VARCHAR(255),
      purchased_count INT DEFAULT 0
    )
  `;

  try {
    await connection.execute(createTableQuery);
    console.log("Table 'purchase_analytics' ensured.");
  } catch (error) {
    console.error("Error ensuring table exists:", error);
    throw error;
  }
};
```

Figure C.21: addPriceAnalytics lambda function - 1

```
export const handler = async (event) => {
  console.log("Event received:", event);

  const { album_id, album_name } = JSON.parse(event.body);

  try {
    const connection = await connectToDB();
    await ensureTableExists(connection);

    const [rows] = await connection.execute(
      "SELECT purchased_count FROM purchase_analytics WHERE album_id = ?",
      [album_id]
    );

    if (rows.length > 0) {
      await connection.execute(
        "UPDATE purchase_analytics SET purchased_count = purchased_count + 1 WHERE album_id = ?",
        [album_id]
      );
    } else {
      await connection.execute(
        "INSERT INTO purchase_analytics (album_id, album_name, purchased_count) VALUES (?, ?, 1)",
        [album_id, album_name]
      );
    }

    await connection.end();

    return {
      statusCode: 200,
      body: JSON.stringify({ message: "Purchase updated successfully" }),
    };
  } catch (error) {
    console.error("Error updating purchase:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ message: "Error updating purchase" }),
    };
  }
};
```

Figure C.22: addPriceAnalytics lambda function - 2

```
export const handler = async (event) => {
  console.log("Event received:", event);

  const { album_id, album_name } = JSON.parse(event.body);

  try {
    const connection = await connectToDB();
    await ensureTableExists(connection);

    const [rows] = await connection.execute(
      "SELECT purchased_count FROM purchase_analytics WHERE album_id = ?",
      [album_id]
    );

    if (rows.length > 0) {
      await connection.execute(
        "UPDATE purchase_analytics SET purchased_count = purchased_count + 1 WHERE album_id = ?",
        [album_id]
      );
    } else {
      await connection.execute(
        "INSERT INTO purchase_analytics (album_id, album_name, purchased_count) VALUES (?, ?, 1)",
        [album_id, album_name]
      );
    }

    await connection.end();

    return {
      statusCode: 200,
      body: JSON.stringify({ message: "Purchase updated successfully" }),
    };
  } catch (error) {
    console.error("Error updating purchase:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ message: "Error updating purchase" }),
    };
  }
};
```

Figure C.23: addPriceAnalytics lambda function - 2

```
import mysql from "mysql2/promise";

const DB_HOST = "cluster-1.czoccuoywyke.eu-west-1.rds.amazonaws.com";
const DB_USER = "admin";
const DB_PASSWORD = "RTX4070super";
const DB_NAME = "analytics";

const connectToDB = async () => {
  return await mysql.createConnection({
    host: DB_HOST,
    user: DB_USER,
    password: DB_PASSWORD,
    database: DB_NAME,
  });
};

export const handler = async (event) => {
  const filterBy = event.queryStringParameters?.filterBy || "Default";

  const connection = await connectToDB();

  try {
    let query = "SELECT * FROM song_engagement ORDER BY user_engagement DESC";


    if (filterBy === "Artist") {
      query = "SELECT * FROM song_engagement ORDER BY artist_name";
    } else if (filterBy === "Album") {
      query = "SELECT * FROM song_engagement ORDER BY album_name";
    } else if (filterBy === "Genre") {
      query = "SELECT * FROM song_engagement ORDER BY genre";
    } else if (filterBy === "Artist-Desc") {
      query = "SELECT * FROM song_engagement ORDER BY artist_name DESC";
    } else if (filterBy === "Album-Desc") {
      query = "SELECT * FROM song_engagement ORDER BY album_name DESC";
    } else if (filterBy === "Genre-Desc") {
      query = "SELECT * FROM song_engagement ORDER BY genre DESC";
    }


    const [rows] = await connection.execute(query);

    await connection.end();
```

Figure C.24: getAnalytics lambda function - 1

```
    const [rows] = await connection.execute(query);

    await connection.end();

    return {
      statusCode: 200,
      body: JSON.stringify(rows),
    };
  } catch (error) {
    console.error("Error fetching data:", error);
    await connection.end();

    return {
      statusCode: 500,
      body: JSON.stringify({ message: "Error fetching data" }),
    };
  }
};
```

Figure C.25: getAnalytics lambda function - 2

```
import mysql from "mysql2/promise";

const DB_HOST = "cluster-1.czoccuoywyke.eu-west-1.rds.amazonaws.com";
const DB_USER = "admin";
const DB_PASSWORD = "RTX4070super";
const DB_NAME = "analytics";


const connectToDB = async () => {
  return await mysql.createConnection({
    host: DB_HOST,
    user: DB_USER,
    password: DB_PASSWORD,
    database: DB_NAME,
  });
};


export const handler = async (event) => {
  const filterBy = event.queryStringParameters?.filterBy || "Default";

  const connection = await connectToDB();

  try {
    let query = "SELECT * FROM purchase_analytics ORDER BY purchased_count DESC";

    if (filterBy === "Album") {
      query = "SELECT * FROM purchase_analytics ORDER BY album_name";
    } else if (filterBy === "Album-Desc") {
      query = "SELECT * FROM purchase_analytics ORDER BY album_name DESC";
    }

    const [rows] = await connection.execute(query);

    await connection.end();

    return {
      statusCode: 200,
      body: JSON.stringify(rows),
    };
  } catch (error) {
    console.error("Error fetching data:", error);
    await connection.end();

    return {
      statusCode: 500,
      body: JSON.stringify({ message: "Error fetching data" }),
    };
```

Figure C.26: getPriceAnalytics lambda function

```
import AWS from "aws-sdk";

const sns = new AWS.SNS();
const dynamodb = new AWS.DynamoDB.DocumentClient();

export const handler = async (event) => {
  let albums = [];
  let songs = [];

  try {
    const albumData = await dynamodb.scan({ TableName: "Albums" }).promise();
    albums = albumData.Items;

    const songData = await dynamodb.scan({ TableName: "Songs" }).promise();
    songs = songData.Items;

    const csvContent = createInventoryCSV(albums, songs);

    await sns
      .publish({
        TopicArn: "arn:aws:sns:eu-west-1:975049978807:inventory-report",
        Message: csvContent,
        Subject: "Dreamstreamer Inventory Report",
      })
      .promise();

    return {
      statusCode: 200,
      body: JSON.stringify({ message: "CSV report sent successfully!" }),
    };
  } catch (error) {
    console.error("Error sending report:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Failed to send report" }),
    };
  }
};
```

Figure C.27: sendInventoryReport lambda function - 1

```
const createInventoryMessage = (albums, songs) => {
  let csv = "Albums,Artist\n";

  albums.forEach((album) => {
    csv += `${album.albumName},${album.artistName}\n`;
  });

  csv += "\nSongs,Album\n";
  songs.forEach((song) => {
    csv += `${song.songName},${song.albumName}\n`;
  });

  return csv;
};
```

Figure C.28: sendInventoryReport lambda function - 2

```
import AWS from "aws-sdk";

AWS.config.update({ region: "eu-west-1" });

const cognito = new AWS.CognitoIdentityServiceProvider();

export const handler = async (event) => {
  const userPoolId = "eu-west-1_pKISgVYlu";

  try {
    const params = {
      UserPoolId: userPoolId,
      Limit: 60,
    };

    let users = [];
    let response;

    do {
      response = await cognito.listUsers(params).promise();
      users = users.concat(response.Users);
      params.PaginationToken = response.PaginationToken;
    } while (response.PaginationToken);

    return {
      statusCode: 200,
      body: JSON.stringify(users),
    };
  } catch (error) {
    console.error("Error fetching users: ", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not fetch users" }),
    };
  }
};
```

Figure C.29: getCustomerProfile lambda function

```
import AWS from "aws-sdk";

AWS.config.update({ region: "eu-west-1" });

const cognito = new AWS.CognitoIdentityServiceProvider();

export const handler = async (event) => {
  const userPoolId = "eu-west-1_pKISgVYlu";
  const { username } = JSON.parse(event.body);

  if (!username) {
    return {
      statusCode: 400,
      body: JSON.stringify({ error: "Username is required" }),
    };
  }

  try {
    const params = {
      UserPoolId: userPoolId,
      Username: username,
    };

    await cognito.adminDeleteUser(params).promise();

    return {
      statusCode: 200,
      body: JSON.stringify({
        message: `User ${username} deleted successfully`,
      }),
    };
  } catch (error) {
    console.error("Error deleting user: ", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not delete user" }),
    };
  }
};
```

Figure C.30: deleteCustomerProfile lambda function

```
import AWS from "aws-sdk";
const cognito = new AWS.CognitoIdentityServiceProvider();

export const handler = async (event) => {
  const email = event.request.userAttributes.email;
  const userPoolId = event.userPoolId;
  console.log(JSON.stringify(event, null, 2));

  try {
    const params = {
      UserPoolId: userPoolId,
      Filter: `email = "${email}"`,
      Limit: 1,
    };

    const data = await cognito.listUsers(params).promise();

    if (data.Users && data.Users.length > 0) {
      throw new Error("A user with this email already exists.");
    }
    console.log(JSON.stringify(event, null, 2));

    return event;
  } catch (error) {
    console.error("Pre-signup check error:", error);

    const customError = new Error(
      "Pre-signup validation failed: " + error.message
    );
    customError.code = "InvalidParameterException";
    throw customError;
  }
};
```

Figure C.31: customerSignupEmailAuth lambda function

Appendix D

Data Fetching Images

```
export function AlbumDetail() {
  const [active, setActive] = useState(null);
  const [albums, setAlbums] = useState([]);
```

Figure D.1: Code snippet of the state used to stored album details

```
useEffect(() => {
  async function fetchAlbums() {
    try {
      const response = await fetch(
        "https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/get-album"
      );
      const data = await response.json();
      setAlbums(data);
    } catch (error) {
      console.error("Error fetching albums:", error);
    }
  }

  fetchAlbums();
}, []);
```

Figure D.2: Code snippet of the useEffect hook

```
import AWS from "aws-sdk";

const dynamoDB = new AWS.DynamoDB.DocumentClient();

export const handler = async (event) => {
  const params = {
    TableName: "Albums",
  };

  try {
    const data = await dynamoDB.scan(params).promise();

    return {
      statusCode: 200,
      body: JSON.stringify(data.Items),
    };
  } catch (error) {
    console.error("Error fetching album names:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not fetch album names" }),
    };
  }
};
```

Figure D.3: Code snippet of the lambda function

```
<ul className="grid grid-cols-1 md:grid-cols-4 items-start gap-4 my-10 mx-8">
  {albums.map((album) => (
    <motion.div
      key={album.albumId}
      layout
      initial={{ opacity: 0 }}
      animate={{ opacity: 1 }}
      exit={{ opacity: 0 }}
      className="bg-white dark:bg-neutral-900 rounded-xl overflow-hidden shadow-xl hover:shadow-2xl cursor-pointer
      hover:scale-105 transition-transform ease-in-out duration-300"
      onClick={() => handleAlbumClick(album)}
    >
      <img
        src={album.albumArtUrl}
        alt={album.albumName}
        className="w-full h-60 object-cover object-top"
      />
      <div className="p-4 text-white">
        <h2 className="text-xl font-semibold">{album.albumName}</h2>
        <p className="text-sm">{album.artistName}</p>
        <p className="text-sm">{album.genre}</p>
      </div>
    </motion.div>
  ))}
</ul>
```

Figure D.4: Code snippet of the HTML code block where data from the DynamoDB table is mapped over and displayed

```
export function SongDetails() {
  const [songs, setSongs] = useState([]);
```

Figure D.5: Code snippet of the state used to stored song details

```
useEffect(() => {
  const fetchSongs = async () => {
    const response = await fetch(
      "https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/get-song"
    );
    const songsData = await response.json();

    const songsWithAlbumArt = await Promise.all(
      songsData.map(async (song) => {
        const albumCoverResponse = await fetch(
          `https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/get-album-cover?albumId=${song.albumId}`
        );
        const albumCoverData = await albumCoverResponse.json();
        return { ...song, albumArtUrl: albumCoverData.albumArtUrl };
      })
    );

    setSongs(songsWithAlbumArt);
    setFilteredSongs(songsWithAlbumArt);
  };

  fetchSongs();
}, []);
```

Figure D.6: Code snippet of the useEffect hook

```
import AWS from "aws-sdk";

const dynamoDB = new AWS.DynamoDB.DocumentClient();

export const handler = async (event) => {
  const params = {
    TableName: "Songs",
  };

  try {
    const data = await dynamoDB.scan(params).promise();
    return {
      statusCode: 200,
      body: JSON.stringify(data.Items),
    };
  } catch (error) {
    console.error("Error fetching songs:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not fetch songs" }),
    };
  }
};
```

Figure D.7: Code snippet of the lambda function

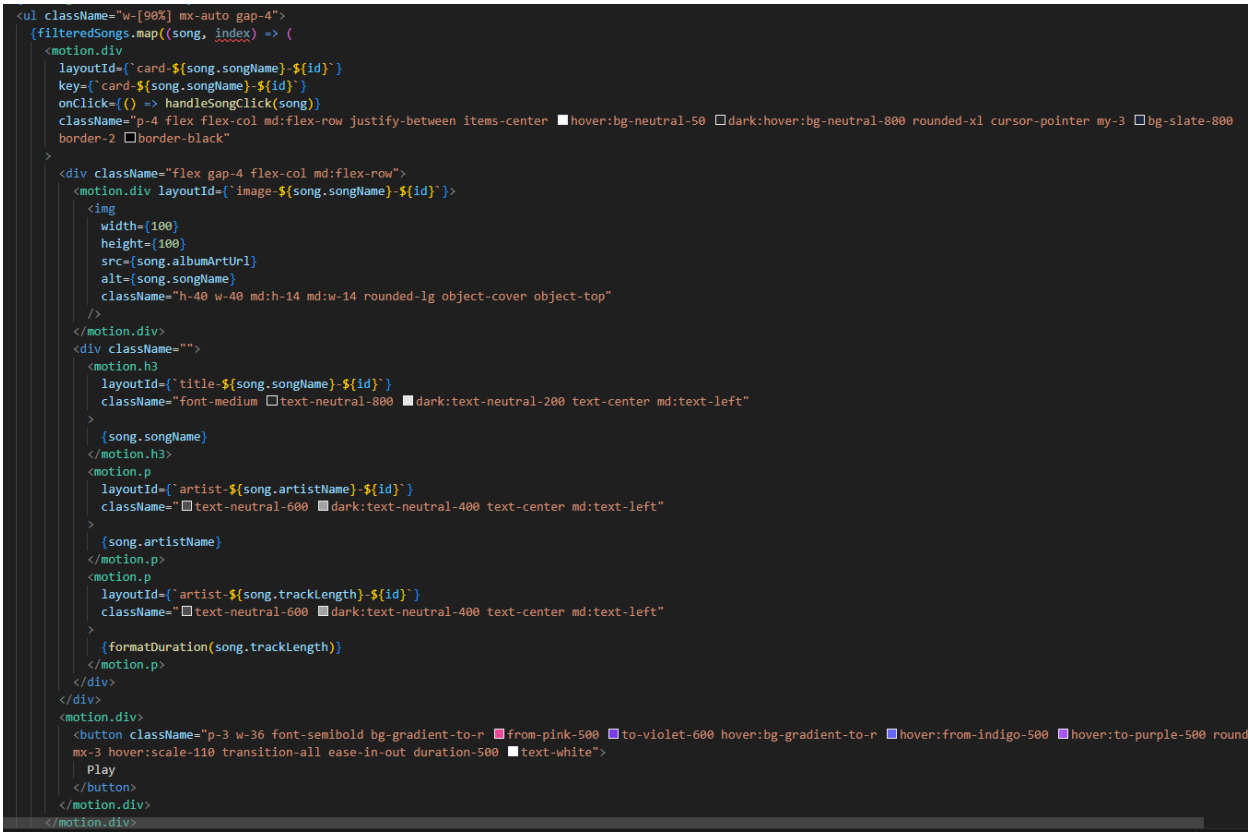


Figure D.8: Code snippet of the HTML code block where data from the DynamoDB table is mapped over and displayed

Appendix E

CRUD Function Images

```
const AddAlbums = () => {
  const [formData, setFormData] = useState({
    albumName: "",
    genre: "",
    artistName: "",
    trackLabel: "",
    bandComposition: "",
    albumYear: "",
  });
  const [albumArt, setAlbumArt] = useState(null);
  const [isLoading, setIsLoading] = useState(false);
```

Figure E.1: addAlbums frontend - 1

```
const handleSubmit = async (e) => {
  e.preventDefault();

  const {
    albumName,
    genre,
    artistName,
    trackLabel,
    bandComposition,
    albumYear,
    albumArtUrl,
  } = formData;

  if (
    !albumName ||
    !genre ||
    !artistName ||
    !trackLabel ||
    !bandComposition ||
    !albumYear ||
    albumArtUrl
  ) {
    alert("Please fill out all the fields before submitting!");
    return;
  }

  if (albumArt) {
    setIsLoading(true);
    try {
      const response = await fetch(
        "https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/add-album",
        {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
          },
          body: JSON.stringify({
            ...formData,
            fileName: albumArt.name,
            fileType: albumArt.type,
          }),
          mode: "cors",
        }
      );
```

Figure E.2: addAlbums frontend - 2

```
      const data = await response.json();

      if (!response.ok) {
        throw new Error(data.error || "Failed to get presigned URL");
      }

      await fetch(data.presignedUrl, {
        method: "PUT",
        headers: {
          "Content-Type": albumArt.type,
        },
        body: albumArt,
      }).then(() => {
        alert("Album added successfully!");
        document.location.reload();
      });
    } catch (error) {
      alert("Failed to submit album: " + error.message);
    } finally {
      setIsLoading(false);
    }
  } else {
    alert("Please select an album art image to upload.");
  }
};
```

Figure E.3: addAlbums frontend - 3

```
import AWS from "aws-sdk";

const dynamoDB = new AWS.DynamoDB.DocumentClient();
const s3 = new AWS.S3();

export const handler = async (event) => {
  const {
    albumName,
    genre,
    artistName,
    trackLabel,
    bandComposition,
    albumYear,
    fileName,
    fileType,
  } = JSON.parse(event.body);

  const albumId = `ALB-ID-${albumName}`;

  const s3Params = {
    Bucket: "dreamstreamer-album-art",
    Key: fileName,
    Expires: 60 * 5,
    ContentType: fileType,
  };

  let uploadUrl = "";

  try {
    uploadUrl = s3.getSignedUrl("putObject", s3Params);
  } catch (error) {
    console.error("Error generating presigned URL:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not generate presigned URL" }),
    };
  }

  const permanentUrl = `https://dreamstreamer-album-art.s3.eu-west-1.amazonaws.com/${fileName}`;

  const createdAt = new Date().toISOString();
```

Figure E.4: addAlbum lambda function - 1

```
  const params = {
    TableName: "Albums",
    Item: {
      albumId: albumId,
      albumName: albumName,
      genre: genre,
      artistName: artistName,
      trackLabel: trackLabel,
      bandComposition: bandComposition,
      albumYear: albumYear,
      albumArtUrl: permanentUrl,
      createdAt: createdAt,
      updatedAt: createdAt,
    },
  };

  try {
    await dynamoDB.put(params).promise();

    return {
      statusCode: 200,
      body: JSON.stringify({
        message: "Album added successfully!",
        presignedUrl: uploadUrl,
      }),
    };
  } catch (error) {
    console.error("Error adding album:", error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not add album" }),
    };
  }
};
```

Figure E.5: addAlbum lambda function - 2

```
const UpdateAlbum = () => {
  const [albums, setAlbums] = useState([]);
  const [selectedAlbum, setSelectedAlbum] = useState(null);
  const [formData, setFormData] = useState({
    albumName: "",
    genre: "",
    artistName: "",
    trackLabel: "",
    bandComposition: "",
    albumYear: "",
  });
  const [albumArt, setAlbumArt] = useState(null);
```

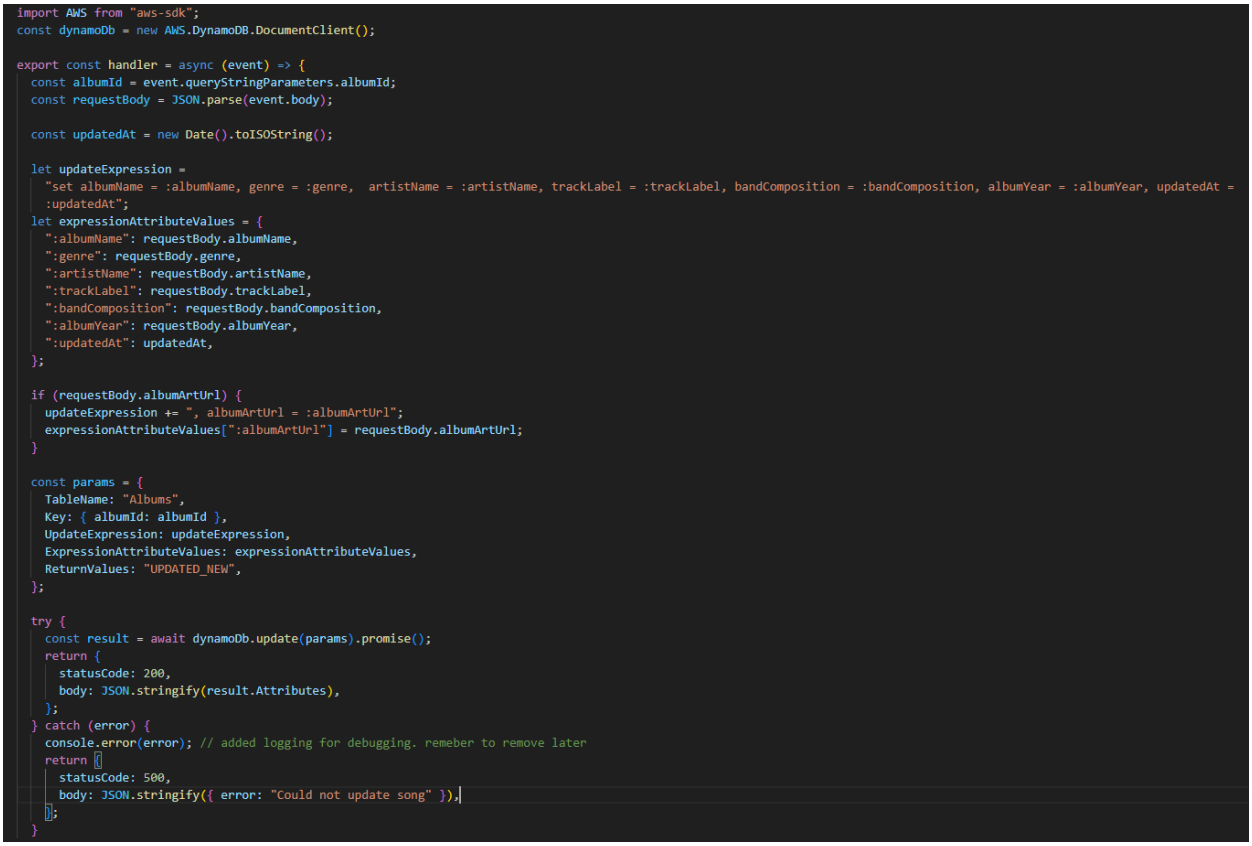
Figure E.6: updateAlbum frontend - 1

```
const handleSubmit = async (e) => {
  e.preventDefault();

  let fileUrl = null;

  if (albumArt) {
    setIsLoading(true);
    try {
      const response = await fetch(
        "https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/upload-new-album-cover",
        {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
          },
          body: JSON.stringify({
            fileName: albumArt.name,
            fileType: albumArt.type,
          }),
        }
      );

      const data = await response.json();

      if (!response.ok || !data.presignedUrl) {
        throw new Error(data.error || "Failed to get presigned URL");
      }

      await fetch(data.presignedUrl, {
        method: "PUT",
        headers: {
          "Content-Type": albumArt.type,
        },
        body: albumArt,
      });

      fileUrl = data.fileUrl;
      console.log("File url: ", fileUrl);
    } catch (error) {
      console.error("Error during file upload:", error);
      return;
    }
  }
```

Figure E.7: updateAlbum frontend - 2

```
  try {
    console.log("Album ID: ", selectedAlbum);
    console.log("File urlz: ", fileUrl);
    await fetch(
      `https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/update-album?albumId=${encodeURIComponent(
        selectedAlbum
      )}`,
      {
        method: "PUT",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          ...formData,
          albumArtUrl: fileUrl || formData.albumArtUrl,
        }),
      }
    ).then(() => {
      alert("Album updated successfully!");
      document.location.reload();
    });
  } catch (error) {
    console.error("Error updating album:", error);
    alert("Failed to update album: " + error.message);
  } finally {
    setIsLoading(false);
  }
};
```

Figure E.8: updateAlbum frontend - 3



```
import AWS from "aws-sdk";
const dynamoDb = new AWS.DynamoDB.DocumentClient();

export const handler = async (event) => {
  const albumId = event.queryStringParameters.albumId;
  const requestBody = JSON.parse(event.body);

  const updatedAt = new Date().toISOString();

  let updateExpression =
    "set albumName = :albumName, genre = :genre,  artistName = :artistName, trackLabel = :trackLabel, bandComposition = :bandComposition, albumYear = :albumYear, updatedAt =
    :updatedAt";
  let expressionAttributeValues = {
    ":albumName": requestBody.albumName,
    ":genre": requestBody.genre,
    ":artistName": requestBody.artistName,
    ":trackLabel": requestBody.trackLabel,
    ":bandComposition": requestBody.bandComposition,
    ":albumYear": requestBody.albumYear,
    ":updatedAt": updatedAt,
  };

  if (requestBody.albumArtUrl) {
    updateExpression += ", albumArtUrl = :albumArtUrl";
    expressionAttributeValues[":albumArtUrl"] = requestBody.albumArtUrl;
  }

  const params = {
    TableName: "Albums",
    Key: { albumId: albumId },
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW",
  };

  try {
    const result = await dynamoDb.update(params).promise();
    return {
      statusCode: 200,
      body: JSON.stringify(result.Attributes),
    };
  } catch (error) {
    console.error(error); // added logging for debugging. remeber to remove later
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not update song" }),
    };
  }
```

Figure E.9: updateAlbum lambda function

```
const handleDelete = async () => {
  if (!albumToDelete) return;

  try {
    const response = await fetch(
      "https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/delete-album",
      {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          albumId: albumToDelete.albumId,
          albumArtUrl: albumToDelete.albumArtUrl,
        }),
      }
    );

    if (!response.ok) {
      throw new Error("Failed to delete album");
    }

    setAlbums((prevAlbums) =>
      prevAlbums.filter((album) => album.albumId !== albumToDelete.albumId)
    );
    setDeleteState(false);
    setAlbumToDelete(null);
    alert("Album deleted successfully!");
  } catch (error) {
    console.error("Error deleting album:", error);
    alert("Failed to delete album: " + error.message);
  }
};

const confirmDelete = (album) => {
  setAlbumToDelete(album);
  setDeleteState(true);
};

const handleFilterChange = (e) => {
  const { name, value } = e.target;
  setFilterOptions((prev) => ({
    ...prev,
    [name]: value,
  }));
};
```

Figure E.10: deleetAlbum frontend

```
import AWS from "aws-sdk";
const dynamoDb = new AWS.DynamoDB.DocumentClient();
const s3 = new AWS.S3();

export const handler = async (event) => {
  const { albumId, albumArtUrl } = JSON.parse(event.body);

  const albumParams = {
    TableName: "Albums",
    Key: {
      albumId: albumId,
    },
  };

  const querySongsParams = {
    TableName: "Songs",
    IndexName: "albumId-songId-index",
    KeyConditionExpression: "albumId = :albumId",
    ExpressionAttributeValues: {
      ":albumId": albumId,
    },
  };

  try {
    await dynamoDb.delete(albumParams).promise();

    const songsData = await dynamoDb.query(querySongsParams).promise();

    if (songsData.Items && songsData.Items.length > 0) {
      const deleteSongPromises = songsData.Items.map((song) => {
        const deleteSongParams = {
          TableName: "Songs",
          Key: {
            songId: song.songId,
          },
        };
        return dynamoDb.delete(deleteSongParams).promise();
      });

      await Promise.all(deleteSongPromises);
    }

    const fileName = albumArtUrl.split("/").pop();
    const s3Params = {
      Bucket: "dreamstreamer-album-art",
      Key: fileName,
    };
```

```
    await s3.deleteObject(s3Params).promise();

    return {
      statusCode: 200,
      body: JSON.stringify({
        message: "Album and associated songs deleted successfully",
      }),

    };
  } catch (error) {
    console.error(error);
    return {
      statusCode: 500,
      body: JSON.stringify({
        error: "Failed to delete album or associated songs",
      }),

    };
  }
};
```

Figure E.12: deleteAlbum lambda function - 1

```
const AddSong = () => {
  const [formData, setFormData] = useState({
    songName: "",
    genre: "",
    albumName: "",
    trackLength: "",
    artistName: "",
    songYear: "",
  });
  const [albums, setAlbums] = useState([]);
  const [songFile, setSongFile] = useState(null);
  const [isLoading, setIsLoading] = useState(false);

  const loadingStates = [···
  ];

  useEffect(() => {
    const fetchAlbums = async () => {
      try {
        const response = await fetch(
          "https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/get-album"
        );
        const data = await response.json();

        if (Array.isArray(data)) {
          const albumNames = data.map((album) => album.albumName);
          setAlbums(albumNames);
          console.log(albums);
        } else {
          setAlbums([]);
        }
      } catch (error) {
        console.error("Error fetching albums:", error);
      }
    };

    fetchAlbums();
  }, []);

  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value,
    });
  };
```

Figure E.13: songAdd frontend - 1

```
const handleChange = (e) => {
  setFormData({
    ...formData,
    [e.target.name]: e.target.value,
  });
};

const handleFileChange = (e) => {
  const file = e.target.files[0];
  setSongFile(file);

  if (file) {
    const audio = new Audio(URL.createObjectURL(file));
    audio.onloadedmetadata = () => {
      const durationInSeconds = Math.floor(audio.duration);
      setFormData({
        ...formData,
        trackLength: durationInSeconds,
      });
    };
  }
};

const handleSubmit = async (e) => {
  e.preventDefault();

  const { songName, genre, albumName, trackLength, artistName, songYear } =
    formData;

  if (
    !songName ||
    !genre ||
    !albumName ||
    !trackLength ||
    !artistName ||
    !songYear ||
    !songFile
  ) {
    alert("Please fill out all fields before submitting.");
    return;
  }

  if (songFile) {
    setIsLoading(true);
    console.log("Submitting form with data:", formData);
    try {
```

Figure E.14: songAdd frontend - 2

```
const handleSubmit = async (e) => {
  if (songFile) {
    setIsLoading(true);
    console.log("Submitting form with data:", formData);
    try {
      const response = await fetch(
        "https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/add-song",
        {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
          },
          body: JSON.stringify({
            ...formData,
            fileName: songFile.name,
            fileType: songFile.type,
          }),
          mode: "cors",
        }
      );

      const data = await response.json();

      console.log("Lambda Response:", data);

      if (!response.ok || !data.presignedUrl) {
        throw new Error(data.error || "Failed to get presigned URL");
      }

      await fetch(data.presignedUrl, {
        method: "PUT",
        headers: {
          "Content-Type": songFile.type,
        },
        body: songFile,
      }).then(() => {
        alert("Added song successfully!");
        document.location.reload();
      });
    } catch (error) {
      console.error("Error during submission:", error);
      alert("Failed to submit song: " + error.message);
    } finally {
      setIsLoading(false);
    }
  } else {
    alert("Please select a song file to upload.");
```

Figure E.15: songAdd frontend - 3

```
import AWS from "aws-sdk";

const dynamoDB = new AWS.DynamoDB.DocumentClient();
const s3 = new AWS.S3();

export const handler = async (event) => {

  const {
    songName,
    genre,
    albumName,
    trackLength,
    artistName,
    songYear,
    fileName,
    fileType,
  } = JSON.parse(event.body);

  console.log("Received albumName:", albumName);

  if (!fileName || !fileType) {
    console.error("fileName or fileType is missing.");
    return {
      statusCode: 400,

      body: JSON.stringify({ error: "File name and type are required." }),
    };
  }

  const songId = `SID-${songName}-${albumName}`;

  console.log("Querying album with name:", albumName);

  let albumId;
  const albumParams = {
    TableName: "Albums",
    IndexName: "albumName-index",
    KeyConditionExpression: "albumName = :albumName",
    ExpressionAttributeValues: {
      ":albumName": albumName,
    },
  };

  try {
    const albumResult = await dynamoDB.query(albumParams).promise();
    albumId =
      albumResult.Items && albumResult.Items.length > 0
        ? albumResult.Items[0].albumId
```

Figure E.16: songAdd lambda - 1

```
    albumResult.Items && albumResult.Items.length > 0
      ? albumResult.Items[0].albumId
      : null;

  if (!albumId) {
    return {
      statusCode: 400,

      body: JSON.stringify({ error: "Album not found." }),
    };
  }
} catch (error) {
  console.error("Error fetching album:", error);
  return {
    statusCode: 500,

    body: JSON.stringify({ error: "Could not fetch album details." }),
  };
}

const s3Params = {
  Bucket: "dreamstreamer-song",
  Key: fileName,
  Expires: 60 * 5,
  ContentType: fileType,
};

let songFileUrl = "";

try {
  songFileUrl = s3.getSignedUrl("putObject", s3Params);
  console.log("Generated presigned URL:", songFileUrl);
} catch (error) {
  console.error("Error generating presigned URL:", error);
  return {
    statusCode: 500,

    body: JSON.stringify({ error: "Could not generate presigned URL" }),
  };
}

const createdAt = new Date().toISOString();
```

Figure E.17: songAdd lambda - 2

```
  const params = {
    TableName: "Songs",
    Item: {
      songId: songId,
      songName: songName,
      genre: genre,
      albumName: albumName,
      albumId: albumId,
      trackLength: trackLength,
      artistName: artistName,
      songYear: songYear,
      songFileUrl: songFileUrl.split("?")[0],
      createdAt: createdAt,
      updatedAt: createdAt,
    },
  };

  try {
    await dynamoDB.put(params).promise();

    return {
      statusCode: 200,

      body: JSON.stringify({
        message: "Song added successfully!",
        presignedUrl: songFileUrl,
      }),
    };
  } catch (error) {
    console.error("Error adding song:", error);
    return {
      statusCode: 500,

      body: JSON.stringify({ error: "Could not add song" }),
    };
  }
};
```

Figure E.18: songAdd lamdba - 3

```
const UpdateSong = () => {
  const [songs, setSongs] = useState([]);
  const [albums, setAlbums] = useState([]);
  const [selectedSong, setSelectedSong] = useState(null);
  const [formData, setFormData] = useState({
    songName: "",
    genre: "",
    albumName: "",
    trackLength: "",
    artistName: "",
    songYear: "",
  });
  const [songFile, setSongFile] = useState(null);
  const [isLoading, setIsLoading] = useState(false);

  const loadingStates = [···
  ];

  useEffect(() => {
    const fetchSongs = async () => {
      try {
        const response = await fetch(
          "https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/get-song"
        );
        const data = await response.json();
        setSongs(data);
        console.log(data);
      } catch (error) {
        console.error("Error fetching songs:", error);
      }
    };

    fetchSongs();
  }, []);

  const handleSongSelection = async (e) => {
    const songId = e.target.value;
    setSelectedSong(songId);
```

Figure E.19: songUpdate frontend - 1

```
  if (songId) {
    try {
      const response = await fetch(
        `https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/get-song-by-id?songId=${encodeURIComponent(
          songId
        )}`
      );
      const song = await response.json();

      setFormData({
        songName: song.songName,
        genre: song.genre,
        albumName: song.albumName,
        trackLength: song.trackLength,
        artistName: song.artistName,
        songYear: song.songYear,
      });
    } catch (error) {
      console.error("Error fetching song details:", error);
    }
  }
};

const handleChange = (e) => {
  setFormData({
    ...formData,
    [e.target.name]: e.target.value,
  });
};

const handleFileChange = (e) => {
  const file = e.target.files[0];
  setSongFile(file);

  if (file) {
    const audio = new Audio(URL.createObjectURL(file));
    audio.onloadedmetadata = () => {
      const durationInSeconds = Math.floor(audio.duration);
      setFormData({
        ...formData,
        trackLength: durationInSeconds,
      });
    };
  }
};
```

Figure E.20: songUpdate frontend - 2

```
const handleSubmit = async (e) => {
  e.preventDefault();

  let fileUrl = null;

  if (songFile) {
    setIsLoading(true);
    try {
      const response = await fetch(
        "https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/upload-new-song",
        {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
          },
          body: JSON.stringify({
            fileName: songFile.name,
            fileType: songFile.type,
          }),
        }
      );

      const data = await response.json();

      if (!response.ok || !data.presignedUrl) {
        throw new Error(data.error || "Failed to get presigned URL");
      }

      await fetch(data.presignedUrl, {
        method: "PUT",
        headers: {
          "Content-Type": songFile.type,
        },
        body: songFile,
      }).then(() => {
        alert("Song details updated successfully!");
        document.location.reload();
      });

      fileUrl = data.fileUrl;
    } catch (error) {
      console.error("Error during file upload:", error);
      return;
    } finally {
      setIsLoading(false);
```

Figure E.21: songUpdate frontend - 3

```
  try {
    console.log("Song ID: ", selectedSong);
    await fetch(
      `https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/update-song?songId=${encodeURIComponent(
        selectedSong
      )}`,
      {
        method: "PUT",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          ...formData,
          songFileUrl: fileUrl || formData.songFileUrl,
        }),
      }
    ).then(() => {
      alert("Song updated successfully!");
      document.location.reload();
    });
  } catch (error) {
    console.error("Error updating song:", error);
    // alert("Failed to update song: " + error.message);
  }
};

useEffect(() => {
  const fetchAlbums = async () => {
    try {
      const response = await fetch(
        "https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/get-album"
      );
      const data = await response.json();

      if (Array.isArray(data)) {
        const albumNames = data.map((album) => album.albumName);
        setAlbums(albumNames);
        console.log(albums);
      } else {
        setAlbums([]);
      }
    } catch (error) {
      console.error("Error fetching albums:", error);
    }
  };

  fetchAlbums();
```

Figure E.22: songUpdate frontend - 4

```
import AWS from "aws-sdk";
const dynamoDb = new AWS.DynamoDB.DocumentClient();

export const handler = async (event) => {
  const songId = event.queryStringParameters.songId;
  const requestBody = JSON.parse(event.body);

  const updatedAt = new Date().toISOString();

  let updateExpression =
    "set songName = :songName, genre = :genre, albumName = :albumName, trackLength = :trackLength, artistName = :artistName, songYear = :songYear, updatedAt = :updatedAt";
  let expressionAttributeValues = {
    ":songName": requestBody.songName,
    ":genre": requestBody.genre,
    ":albumName": requestBody.albumName,
    ":trackLength": requestBody.trackLength,
    ":artistName": requestBody.artistName,
    ":songYear": requestBody.songYear,
    ":updatedAt": updatedAt,
  };

  if (requestBody.songFileUrl) {
    updateExpression += ", songFileUrl = :songFileUrl";
    expressionAttributeValues[":songFileUrl"] = requestBody.songFileUrl;
  }

  const params = {
    TableName: "Songs",
    Key: { songId: songId },
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW",
  };

  try {
    const result = await dynamoDb.update(params).promise();
    return {
      statusCode: 200,
      body: JSON.stringify(result.Attributes),
    };
  } catch (error) {
    console.error(error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Could not update song" }),
    };
  }
};
```

Figure E.23: songUpdate lambda

```
const DeleteSong = () => {
  const [songs, setSongs] = useState([]);
  const [filteredSongs, setFilteredSongs] = useState([]);
  const [deleteState, setDeleteState] = useState(false);
  const [songToDelete, setSongToDelete] = useState(null);
  const [filterOptions, setFilterOptions] = useState({
    album: "",
    genre: "",
    artist: "",
  });

  useEffect(() => {
    const fetchSongs = async () => {
      const response = await fetch(
        "https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/get-song"
      );
      const songsData = await response.json();

      setSongs(songsData);
      setFilteredSongs(songsData);
    };

    fetchSongs();
  }, []);

  useEffect(() => {
    const filtered = songs.filter((song) => {
      return (
        (filterOptions.album === "" ||
          song.albumName === filterOptions.album) &&
        (filterOptions.genre === "" || song.genre === filterOptions.genre) &&
        (filterOptions.artist === "" ||
          song.artistName === filterOptions.artist)
      );
    });

    setFilteredSongs(filtered);
  }, [filterOptions, songs]);
```

Figure E.24: songDelete frontend - 1

```
const handleDelete = async () => {
  if (!songToDelete) return;

  try {
    const response = await fetch(
      "https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/delete-song",
      {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          songId: songToDelete.songId,
          songFileUrl: songToDelete.songFileUrl,
        }),
      }
    );

    if (!response.ok) {
      throw new Error("Failed to delete song");
    }

    setSongs((prevSongs) =>
      prevSongs.filter((song) => song.songId !== songToDelete.songId)
    );
    setDeleteState(false);
    setSongToDelete(null);
    alert("Song deleted successfully!");
  } catch (error) {
    console.error("Error deleting song:", error);
    alert("Failed to delete song: " + error.message);
  }
};

const confirmDelete = (song) => {
  setSongToDelete(song);
  setDeleteState(true);
};

const handleFilterChange = (e) => {
  const { name, value } = e.target;
  setFilterOptions((prev) => ({
    ...prev,
    [name]: value,
  }));
};
```

Figure E.25: songDelete frontend - 2

```
import AWS from "aws-sdk";
const dynamoDb = new AWS.DynamoDB.DocumentClient();
const s3 = new AWS.S3();

export const handler = async (event) => {
  const { songId, songFileUrl } = JSON.parse(event.body);

  const params = {
    TableName: "Songs",
    Key: {
      songId: songId,
    },
  };

  try {
    await dynamoDb.delete(params).promise();

    const fileName = songFileUrl.split("/").pop();

    const s3Params = {
      Bucket: "dreamstreamer-song",
      Key: fileName,
    };

    await s3.deleteObject(s3Params).promise();

    return {
      statusCode: 200,
      body: JSON.stringify({ message: "Song deleted successfully" }),
    };
  } catch (error) {
    console.error(error);
    return {
      statusCode: 500,
      body: JSON.stringify({ error: "Failed to delete song" }),
    };
  }
};
```

Figure E.26: songDelete lambda

Appendix F

Filter Logic

```
const [songs, setSongs] = useState([]);
const [filteredSongs, setFilteredSongs] = useState([]);
const [active, setActive] = useState(null);
const [filters, setFilters] = useState({ album: "", genre: "", artist: "" });
```

Figure F.1: Filter frontend logic

```
useEffect(() => {
  const fetchSongs = async () => {
    const response = await fetch(
      "https://k9wnoczy6f.execute-api.eu-west-1.amazonaws.com/dev/get-song"
    );
    const songsData = await response.json();

    const songsWithAlbumArt = await Promise.all(
      songsData.map(async (song) => {
        const albumCoverResponse = await fetch(
          `https://vud6lh3r68.execute-api.eu-west-1.amazonaws.com/dev/get-album-cover?albumId=${song.albumId}`
        );
        const albumCoverData = await albumCoverResponse.json();
        return { ...song, albumArtUrl: albumCoverData.albumArtUrl };
      })
    );

    setSongs(songsWithAlbumArt);
    setFilteredSongs(songsWithAlbumArt);
  };

  fetchSongs();
}, []);
```

Figure F.2: Filter frontend logic

```
useEffect(() => {
  const filtered = songs.filter((song) => {
    return (
      (filters.album === "" || song.albumName === filters.album) &&
      (filters.genre === "" || song.genre === filters.genre) &&
      (filters.artist === "" || song.artistName === filters.artist)
    );
  });
  setFilteredSongs(filtered);
}, [filters, songs]);
```

Figure F.3: Filter frontend logic

```
const handleFilterChange = (e) => {
  const { name, value } = e.target;
  setFilters((prevFilters) => ({
    ...prevFilters,
    [name]: value,
  }));
};
```

Figure F.4: Filter frontend logic

```
const uniqueAlbums = [...new Set(songs.map((song) => song.albumName))];
const uniqueGenres = [...new Set(songs.map((song) => song.genre))];
const uniqueArtists = [...new Set(songs.map((song) => song.artistName))];
```

Figure F.5: Filter frontend logic

```jsx
<div className="flex gap-4 my-4 ms-24">
  <select
    name="album"
    value={filters.album}
    onChange={handleFilterChange}
    className="p-2 border rounded-lg"
  >
    <option value="">All Albums</option>
    {uniqueAlbums.map((album) => (
      <option key={album} value={album}>
        {album}
      </option>
    ))}
  </select>

  <select
    name="genre"
    value={filters.genre}
    onChange={handleFilterChange}
    className="p-2 border rounded-lg"
  >
    <option value="">All Genres</option>
    {uniqueGenres.map((genre) => (
      <option key={genre} value={genre}>
        {genre}
      </option>
    ))}
  </select>

  <select
    name="artist"
```

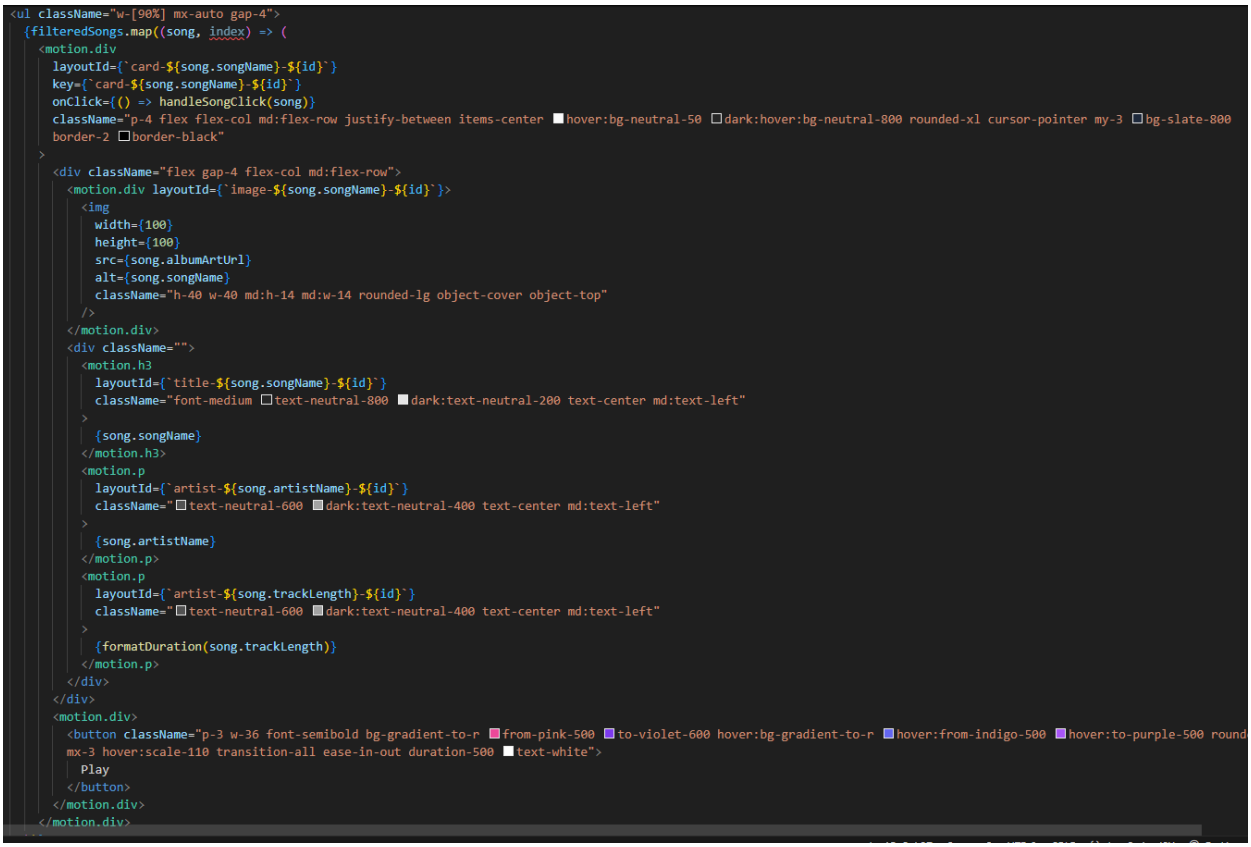


Figure F.7: Filter frontend logic

Appendix G

Analytics Images

```
const handleSongClick = async (song) => {
  setActive(song);

  const payload = {
    album_id: song.albumId,
    song_id: song.songId,
    album_name: song.albumName,
    song_name: song.songName,
    genre: song.genre,
    artist_name: song.artistName,
    track_length: song.trackLength,
  };

  try {
    console.log("Payload", JSON.stringify(payload));
    const response = await fetch(
      "https://dfg5xvb41m.execute-api.eu-west-1.amazonaws.com/dev/update-engagement",
      {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify(payload),
      }
    );

    if (response.ok) {
      console.log("Engagement updated successfully!");
    } else {
      console.error("Failed to update engagement.");
    }
  } catch (error) {
    console.error("Error:", error);
```

Figure G.1: handleSongClick Function

```
import mysql from "mysql2/promise";

const DB_HOST = "cluster-1.czoccuoywyke.eu-west-1.rds.amazonaws.com";
const DB_USER = "admin";
const DB_PASSWORD = "RTX4070super";
const DB_NAME = "analytics";

const connectToDB = async () => {
  return await mysql.createConnection({
    host: DB_HOST,
    user: DB_USER,
    password: DB_PASSWORD,
    database: DB_NAME,
  });
};

const ensureTableExists = async (connection) => {
  const createTableQuery = `
    CREATE TABLE IF NOT EXISTS song_engagement (
      id INT AUTO_INCREMENT PRIMARY KEY,
      album_id VARCHAR(255),
      song_id VARCHAR(255) NOT NULL,
      album_name VARCHAR(255),
      song_name VARCHAR(255),
      genre VARCHAR(255),
      artist_name VARCHAR(255),
      track_length INT,
      user_engagement INT DEFAULT 0,
      UNIQUE(song_id)
    )
  `;

  try {
    await connection.execute(createTableQuery);
    console.log("Table 'song_engagement' ensured.");
  } catch (error) {
    console.error("Error ensuring table exists:", error);
```

Figure G.2: addAnalytics lambda function - 1

```
export const handler = async (event) => {
  console.log("Event received:", event);


  const { album_id, song_id, album_name, song_name, genre, artist_name, track_length } = JSON.parse(event.body);


  try {
    const connection = await connectToDB();
    await ensureTableExists(connection);

    const [rows] = await connection.execute(
      "SELECT user_engagement FROM song_engagement WHERE song_id = ?",
      [song_id]
    );

    if (rows.length > 0) {
      await connection.execute(
        "UPDATE song_engagement SET user_engagement = user_engagement + 1 WHERE song_id = ?",
        [song_id]
      );
    } else {
      await connection.execute(
        "INSERT INTO song_engagement (album_id, song_id, album_name, song_name, genre, artist_name, track_length, user_engagement) VALUES (?, ?, ?, ?, ?, ?, ?, 1)",
        [album_id, song_id, album_name, song_name, genre, artist_name, track_length]
      );
    }

    await connection.end();

    return {
      statusCode: 200,
      body: JSON.stringify({ message: "Engagement updated successfully" }),
```

Figure G.3: addAnalytics lambda function - 2

```
const handlePurchaseClick = async () => {
  const payload = {
    album_id: active.albumId,
    album_name: active.albumName,
  };

  try {
    const response = await fetch(
      "https://dfg5xvb41m.execute-api.eu-west-1.amazonaws.com/dev/update-price-analytics",
      {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify(payload),
      }
    );

    if (response.ok) {
      alert("Purchase recorded successfully!");
    } else {
      console.error("Failed to record purchase.");
      alert("Failed to record purchase. Please try again.");
    }
  } catch (error) {
    console.error("Error:", error);
    alert("An error occurred. Please try again.");
  }
};
```

Figure G.4: handlePurchaseAnalytics Function

```
import mysql from 'mysql2/promise';

const DB_HOST = 'cluster-1.czoccuoywyke.eu-west-1.rds.amazonaws.com';
const DB_USER = 'admin';
const DB_PASSWORD = 'RTX4070super';
const DB_NAME = 'analytics';

const connectToDB = async () => {
  return await mysql.createConnection({
    host: DB_HOST,
    user: DB_USER,
    password: DB_PASSWORD,
    database: DB_NAME,
  });
};

const ensureTableExists = async (connection) => {
  const createTableQuery = `
    CREATE TABLE IF NOT EXISTS purchase_analytics (
      album_id VARCHAR(255) PRIMARY KEY,
      album_name VARCHAR(255),
      purchased_count INT DEFAULT 0
    )
  `;

  try {
    await connection.execute(createTableQuery);
    console.log("Table 'purchase_analytics' ensured.");
  } catch (error) {
    console.error("Error ensuring table exists:", error);
    throw error;
  }
};

export const handler = async (event) => {
  console.log("Event received:", event);
```

Figure G.5: addPurchaseAnalytics lambda function - 1

```
export const handler = async (event) => {
  console.log("Event received:", event);

  const { album_id, album_name } = JSON.parse(event.body);

  try {
    const connection = await connectToDB();
    await ensureTableExists(connection);

    const [rows] = await connection.execute(
      "SELECT purchased_count FROM purchase_analytics WHERE album_id = ?",
      [album_id]
    );

    if (rows.length > 0) {
      // Update the purchased count if the album already exists
      await connection.execute(
        "UPDATE purchase_analytics SET purchased_count = purchased_count + 1 WHERE album_id = ?",
        [album_id]
      );
    } else {
      // Insert new record if the album does not exist
      await connection.execute(
        "INSERT INTO purchase_analytics (album_id, album_name, purchased_count) VALUES (?, ?, 1)",
        [album_id, album_name]
      );
    }

    await connection.end();

    return {
      statusCode: 200,
      body: JSON.stringify({ message: "Purchase updated successfully" }),
```

Figure G.6: addPurchaseAnalytics lambda function - 2

```
const AnalyticsOverview = () => {
  const [songs, setSongs] = useState([]);
  const [filter, setFilter] = useState("Default");

  useEffect(() => {
    const fetchSongs = async () => {
      try {
        const response = await fetch(
          `https://dfg5xvb41m.execute-api.eu-west-1.amazonaws.com/dev/get-analytics?filterBy=${filter}`
        );
        const data = await response.json();
        setSongs(data);
      } catch (error) {
        console.error("Error fetching songs:", error);
      }
    };

    fetchSongs();
  }, [filter]);
```

Figure G.7: getAnalytics frontend

```
import mysql from "mysql2/promise";

const DB_HOST = "cluster-1.czoccuoywyke.eu-west-1.rds.amazonaws.com";
const DB_USER = "admin";
const DB_PASSWORD = "RTX4070super";
const DB_NAME = "analytics";

// Function to connect to RDS
const connectToDB = async () => {
  return await mysql.createConnection({
    host: DB_HOST,
    user: DB_USER,
    password: DB_PASSWORD,
    database: DB_NAME,
  });
};

// Lambda handler function
export const handler = async (event) => {
  // Get filter from query string parameters
  const filterBy = event.queryStringParameters?.filterBy || "Default";

  const connection = await connectToDB();

  try {
    let query = "SELECT * FROM song_engagement ORDER BY user_engagement DESC";

    // Apply filtering based on user input
    if (filterBy === "Artist") {
      query = "SELECT * FROM song_engagement ORDER BY artist_name";
    } else if (filterBy === "Album") {
      query = "SELECT * FROM song_engagement ORDER BY album_name";
    } else if (filterBy === "Genre") {
      query = "SELECT * FROM song_engagement ORDER BY genre";
    } else if (filterBy === "Artist-Desc") {
      query = "SELECT * FROM song_engagement ORDER BY artist_name DESC";
    } else if (filterBy === "Album-Desc") {
```

Figure G.8: getAnalytics lambda function - 1

```
try {
  let query = "SELECT * FROM song_engagement ORDER BY user_engagement DESC";

  if (filterBy === "Artist") {
    query = "SELECT * FROM song_engagement ORDER BY artist_name";
  } else if (filterBy === "Album") {
    query = "SELECT * FROM song_engagement ORDER BY album_name";
  } else if (filterBy === "Genre") {
    query = "SELECT * FROM song_engagement ORDER BY genre";
  } else if (filterBy === "Artist-Desc") {
    query = "SELECT * FROM song_engagement ORDER BY artist_name DESC";
  } else if (filterBy === "Album-Desc") {
    query = "SELECT * FROM song_engagement ORDER BY album_name DESC";
  } else if (filterBy === "Genre-Desc") {
    query = "SELECT * FROM song_engagement ORDER BY genre DESC";
  }


  const [rows] = await connection.execute(query);

  // Close the connection after execution
  await connection.end();

  return {
    statusCode: 200,
    body: JSON.stringify(rows),
    headers: {
      "Access-Control-Allow-Origin": "*",
      "Access-Control-Allow-Headers": "Content-Type",
      "Access-Control-Allow-Methods": "GET",
    },
  };
} catch (error) {
```

Figure G.9: getAnalytics lambda function - 2

```
const PurchaseAnalytics = () => {
  const [albums, setAlbums] = useState([]);
  const [filter, setFilter] = useState("Default");

  useEffect(() => {
    const fetchAlbums = async () => {
      try {
        const response = await fetch(
          `https://dfg5xvb41m.execute-api.eu-west-1.amazonaws.com/dev/get-price-analytics?filterBy=${filter}`
        );
        const data = await response.json();
        setAlbums(data);
      } catch (error) {
        console.error("Error fetching songs:", error);
      }
    };

    fetchAlbums();
  }, [filter]);
```

Figure G.10: getPriceAnalytics frontend

```
import mysql from "mysql2/promise";

const DB_HOST = "cluster-1.czoccuoywyke.eu-west-1.rds.amazonaws.com";
const DB_USER = "admin";
const DB_PASSWORD = "RTX4070super";
const DB_NAME = "analytics";


const connectToDB = async () => {
  return await mysql.createConnection({
    host: DB_HOST,
    user: DB_USER,
    password: DB_PASSWORD,
    database: DB_NAME,
  });
};


export const handler = async (event) => {
  const filterBy = event.queryStringParameters?.filterBy || "Default";

  const connection = await connectToDB();

  try {
    let query = "SELECT * FROM purchase_analytics ORDER BY purchased_count DESC";


    if (filterBy === "Album") {
      query = "SELECT * FROM purchase_analytics ORDER BY album_name";
```

Figure G.11: getPriceAnalytics lambda function - 1

```
  if (filterBy === "Album") {
    query = "SELECT * FROM purchase_analytics ORDER BY album_name";
  } else if (filterBy === "Album-Desc") {
    query = "SELECT * FROM purchase_analytics ORDER BY album_name DESC";
  }

  const [rows] = await connection.execute(query);


  await connection.end();

  return {
    statusCode: 200,
    body: JSON.stringify(rows),
    headers: {
      "Access-Control-Allow-Origin": "*",
      "Access-Control-Allow-Headers": "Content-Type",
      "Access-Control-Allow-Methods": "GET",
    },
  };
} catch (error) {
  console.error("Error fetching data:", error);
  await connection.end();
```

Figure G.12: getPriceAnalytics lambda function - 2

Appendix H

Cognito Authentication

```
const poolData = {
  UserPoolId: "eu-west-1_pKISgVYlu",
  ClientId: "6sd5qel0768tt39pst5991pf23",
};
const userPool = new CognitoUserPool(poolData);

const SignUp = () => {
  const [username, setUsername] = useState("");
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [address, setAddress] = useState("");
  const [phoneNumber, setPhoneNumber] = useState("");
  const [message, setMessage] = useState("");
  const [isSignedUp, setIsSignedUp] = useState(false);
  const [confirmationCode, setConfirmationCode] = useState("");

  // Signup function
  const handleSignUp = (e) => {
    e.preventDefault();

    const attributeList = [
      new CognitoUserAttribute({ Name: "email", Value: email }),
      new CognitoUserAttribute({ Name: "name", Value: name }),
      new CognitoUserAttribute({ Name: "address", Value: address }),
      new CognitoUserAttribute({ Name: "phone_number", Value: phoneNumber }),
      new CognitoUserAttribute({ Name: "custom:user_type", Value: "customer" }),
    ];

    console.log("Signing up user:", username);
```

Figure H.1: Cognito signup code

```
userPool.signUp(
  username,
  password,
  attributeList,
  null,
  (err, result) => {
    if (err) {
      if (err.code === "UsernameExistsException") {
        setMessage(
          "This username is already taken. Please choose another."
        );
      } else if (
        err.code === "InvalidParameterException" &&
        err.message.includes("email")
      ) {
        setMessage(
          "This email is already registered. Please use a different email."
        );
      } else {
        setMessage(`Error: ${err.message || JSON.stringify(err)}`);
      }
      console.error("Sign up error:", err);
      return;
    }

    console.log("Sign up result:", result);
    setMessage(
      "Sign-up successful! Please check your email for the confirmation code."
    );
    setIsSignedUp(true);
  }
);
```

Figure H.2: Cognito signup code - 2

```
const handleConfirmation = (e) => {
  e.preventDefault();

  const userData = {
    Username: username,
    Pool: userPool,
  };

  const cognitoUser = new CognitoUser(userData);
  cognitoUser.confirmRegistration(confirmationCode, true, (err, result) => {
    if (err) {
      setMessage(`Error: ${err.message || JSON.stringify(err)}`);
      return;
    }
    setMessage("Account confirmed successfully! You can now log in.");
  });
};
```

Figure H.3: Cognito signup code - 3

```
const poolData = {
  UserPoolId: "eu-west-1_pKISgVYlu",
  ClientId: "6sd5qel0768tt39pst5991pf23",
};
const userPool = new CognitoUserPool(poolData);

const Login = () => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");
  const [message, setMessage] = useState("");
  const navigate = useNavigate();
  const { login } = useContext(AuthContext);

  const handleLogin = (e) => {
    e.preventDefault();

    const authenticationData = {
      Username: username,
      Password: password,
    };

    const authenticationDetails = new AuthenticationDetails(authenticationData);

    const userData = {
      Username: username,
      Pool: userPool,
    };
    const cognitoUser = new CognitoUser(userData);
```

Figure H.4: Cognito login code - 1

```
  const cognitoUser = new CognitoUser(userData);

  cognitoUser.authenticateUser(authenticationDetails, {
    onSuccess: (result) => {
      const accessToken = result.getAccessToken().getJwtToken();
      setMessage("Login successful!");
      console.log("Login successful:", accessToken);

      login(cognitoUser);

      navigate("/songs");
    },
    onFailure: (err) => {
      setMessage(`Login failed: ${err.message || JSON.stringify(err)}`);
    },
  });
};
```

Figure H.5: Cognito login code - 2

```
const Login = () => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");
  const [error, setError] = useState("");
  const { login } = useContext(AuthContext);
  const navigate = useNavigate();

  const handleSubmit = (e) => {
    e.preventDefault();
    login(username, password, (err) => {
      if (err) {
        setError(err.message);
      } else {
        navigate("/");
      }
    });
  };
```

Figure H.6: Cognito admin login code

```
import React, { createContext, useState, useEffect } from "react";
import { CognitoUserPool } from "amazon-cognito-identity-js";
import { poolData } from "./config";

const UserPool = new CognitoUserPool(poolData);

export const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);
  const [email, setEmail] = useState(null);
  const [username, setUsername] = useState(null);
  const [name, setName] = useState(null);
  const [userType, setUserType] = useState(null);
  const [address, setAddress] = useState(null);
  const [phoneNumber, setPhoneNumber] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const loggedInUser = UserPool.getCurrentUser();
    if (loggedInUser) {
      loggedInUser.getSession((err, session) => {
        if (session && session.isValid()) {
          setUser(loggedInUser);
          setUsername(loggedInUser.getUsername());

          loggedInUser.getUserAttributes((err, attributes) => {
            if (err) {
              console.error("Failed to get user attributes", err);
            } else {
              const emailAttr = attributes.find(
                (attr) => attr.Name === "email"
              );
              const nameAttr = attributes.find((attr) => attr.Name === "name");
              const userTypeAttr = attributes.find(
                (attr) => attr.Name === "custom:user_type"
```

Figure H.7: Authcontext code

```
            const nameAttr = attributes.find((attr) => attr.Name === "name");
            const userTypeAttr = attributes.find(
              (attr) => attr.Name === "custom:user_type"
            );
            const addressAttr = attributes.find(
              (attr) => attr.Name === "address"
            );
            const phoneAttr = attributes.find(
              (attr) => attr.Name === "phone_number"
            );

            if (emailAttr) setEmail(emailAttr.Value);
            if (nameAttr) setName(nameAttr.Value);
            if (userTypeAttr) setUserType(userTypeAttr.Value);
            if (addressAttr) setAddress(addressAttr.Value);
            if (phoneAttr) setPhoneNumber(phoneAttr.Value);
          }
        });
      }
      setLoading(false);
    });
  } else {
    setLoading(false);
  }
}, []);
```

Figure H.8: Authcontext code - 2

```
const login = (user) => {
  setUser(user);
  setUsername(user.getUsername());

  user.getUserAttributes((err, attributes) => {
    if (err) {
      console.error("Failed to get user attributes", err);
    } else {
      const emailAttr = attributes.find((attr) => attr.Name === "email");
      const nameAttr = attributes.find((attr) => attr.Name === "name");
      const userTypeAttr = attributes.find(
        (attr) => attr.Name === "custom:user_type"
      );
      const addressAttr = attributes.find((attr) => attr.Name === "address");
      const phoneAttr = attributes.find(
        (attr) => attr.Name === "phone_number"
      );

      if (emailAttr) setEmail(emailAttr.Value);
      if (nameAttr) setName(nameAttr.Value);
      if (userTypeAttr) setUserType(userTypeAttr.Value);
      if (addressAttr) setAddress(addressAttr.Value);
      if (phoneAttr) setPhoneNumber(phoneAttr.Value);
    }
  });
};
```

Figure H.9: Authcontext code - 3

```
  const logout = () => {
    const loggedInUser = UserPool.getCurrentUser();
    if (loggedInUser) {
      loggedInUser.signOut();
      setUser(null);
      setEmail(null);
      setName(null);
      setUsername(null);
      setUserType(null);
      setAddress(null);
      setPhoneNumber(null);
    }
  };

  return (
    <AuthContext.Provider
      value={{
        user,
        username,
        name,
        email,
        userType,
        address,
        phoneNumber,
        login,
        logout,
        loading,
      }}
    >
      {children}
    </AuthContext.Provider>
  );
};
```

Figure H.10: Authcontext code - 4

```jsx
import React, { useContext } from "react";
import { Navigate } from "react-router-dom";
import { AuthContext } from "../AuthContext";

const ProtectedRoute = ({ children }) => {
  const { user, loading } = useContext(AuthContext);

  if (loading) {
    return <div>Loading...</div>;
  }

  if (!user) {
    return <Navigate to="/login" replace />;
  }

  return children;
};

export default ProtectedRoute;
```

Figure H.11: ProtectedRoute code

Appendix I

Deployment and Hosting

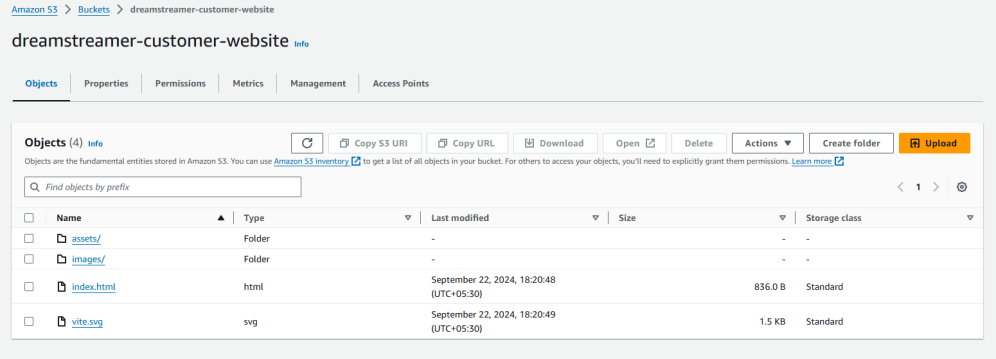


Figure I.1: S3 hosting customer website

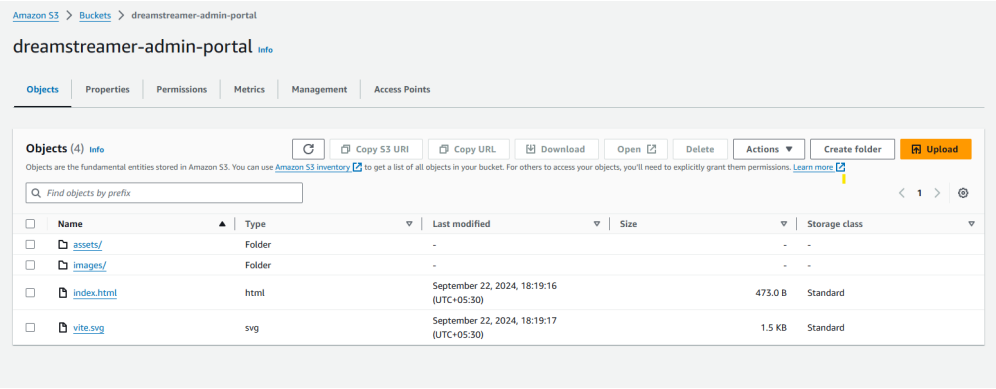


Figure I.2: S3 hosting admin portal website

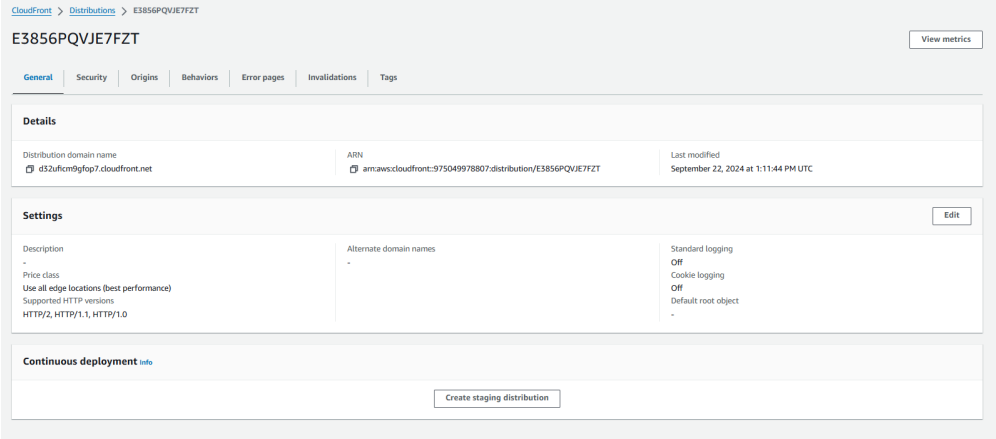


Figure I.3: CloudFront distribution of customer website

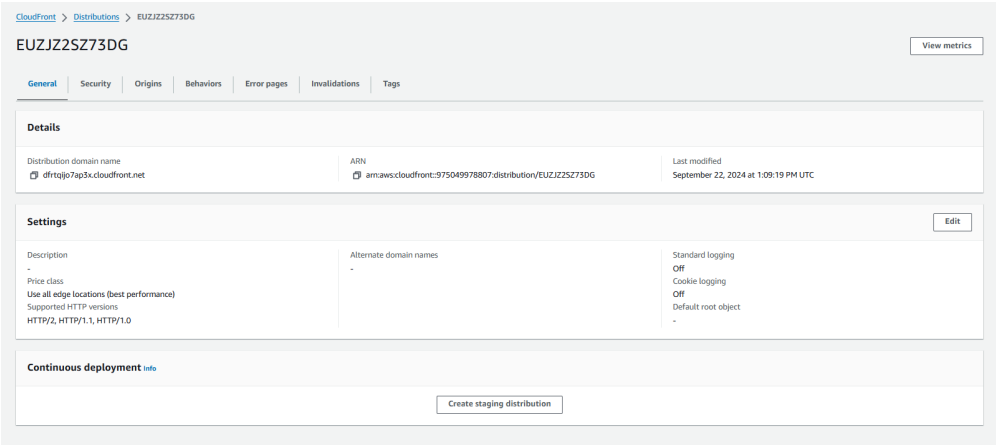


Figure I.4: CloudFront distribution of admin portal website