23CSCI37H

Introduction to Cloud Computing

Final Phase

ID	Name	Email	Contribution
211392	Ahmed Sameh	Ahmed211392@bue.edu.eg	All except Subscription
211896	Abdulrahman Ayman	Abdulrahman211896@bue.edu.eg	All except Subscription
212071	Sara Anabtawi	Sara212071@bue.edu.eg	All except Subscription
201908	Abdelrahman Essam	Abdelrahman201908@bue.edu.eg	Subscription

GitHub repository: https://github.com/Sara212071/FinalCloud.git

Sara 212071, Ahmed 211392 and Abdulrahman 211896 Part---

Admin:

Controller: The code defines an adminController object with asynchronous methods to handle various administrative tasks, interacting with an adminService. In order to create an admin account, a POST request is processed by the createAdmin method, which then calls the relevant service method after retrieving the username and password from the request body. Like this, POST and DELETE requests are handled by methods like sendNotification, suspendUser, banUser, suspendArtist, and banArtist to send notifications or manage suspensions or bans for users and artists. Every method uses the adminService to carry out the underlying business logic and responds with the appropriate status code and message. The controller isolates route handling code from business logic contained in services, adhering to a standard design paradigm in Express.js applications.

Model: Using the Mongoose library, the code defines a MongoDB schema and model for an admin entity. 'Mongoose' is used to import the model and schema first. After that, the adminSchema is built with two mandatory String fields, namely username and password. The 'Admin' name and the defined adminSchema are then associated with the AdminModel, which is built using the model method. The 'admins' MongoDB collection can be accessed through this model, which acts as an interface to perform actions like as creating, editing, and querying admin documents. Lastly, the AdminModel is exported so that it may be utilised in other areas of the application and offers an organised method of managing administrator-related data in the MongoDB database.

Routes: The code creates an Express.js router in 'admin.js' that manages routes for administrative purposes. It imports the adminController and the Express Router from '../Controllers/Admin' along with other required dependencies. The router is set up with multiple routes: another POST route for admin activities like sending alerts, banning and suspending users, and carrying out similar actions on artists; and a POST route for admin authentication via the createAdmin function. The appropriate method in the adminController is linked to each route. By isolating route declarations in a separate file, this modular architecture follows Express.js principles and promotes separation of responsibilities, improving code organisation and maintainability. After that, the router is exported for usage in more general applications.

Services: The code that is given defines an adminService module, which contains the business logic for all of the application's administrative actions. Admins, Users, and Artists can communicate with MongoDB models through this service. By instantiating an AdminModel and storing it in the database, the createAdmin function generates a new administrator. Users and artists receive notifications when the sendNotification method updates their models with supplied data. The methods suspendUser, banUser, suspendArtist, and banArtist alter the associated model fields and save the modifications to the database when they suspend or ban users or artists. Error handling is part of the service to handle situations in which a user is not located while suspended or other unforeseen difficulties arise.

Artist:

Controllers: The provided code defines an adminService module that houses the business logic for all of the administrative actions in the application. Through this service, MongoDB models can be communicated with by Admins, Users, and Artists. The createAdmin function creates a new administrator by instantiating an

AdminModel and saving it in the database. Notifications are sent to users and artists when the sendNotification method updates their models with input data. When they suspend or prohibit users or artists, the methods suspendUser, banUser, suspendArtist, and banArtist modify the related model fields and store the changes in the database. The service includes error handling to deal with circumstances when a user cannot be found while suspended or other unanticipated issues crop up.

Model: A MongoDB schema and model for the representation of artists in a Node.js application utilising Mongoose are defined in the 'Artist.js' file. The ArtistSchema contains all of the necessary fields, such as arrays of object IDs referencing the 'Track' and 'useralbum' models and the artist's name, which is a needed string. These arrays create connections, showing that a single artist can be connected to several singles and albums. 'Artist' name and the specified schema are associated with the ArtistModel that is produced by utilising Mongoose's model function. This model acts as an interface between the application and the 'artists' MongoDB collection, allowing CRUD actions to be carried out on artist documents. In the context of MongoDB data modelling with Mongoose, the module exports the ArtistModel for usage throughout the application, following a modular and maintainable design pattern.

Routes: The 'ArtistRoutes.js' file sets up the 'artistRouter,' an Express.js router, to handle artist-related HTTP requests. From the appropriate directories, it imports the artistController and the Express Router. There are two routes defined: a POST route at the same path that calls the addArtist method from the artistController to add a new artist to the database, and a GET route at the root path ('/') that calls the getArtists method from the artistController to collect all artists. This modular strategy adheres to the rules of Express.js, keeping route definitions and controller logic clearly apart, which makes code organisation easier and improves the maintainability of the programme. After that, the 'artistRouter' is exported for usage in other application components.

Services: The 'ArtistService.js' module is a Node.js application that contains the business logic for artist-related operations. The 'ArtistModel' is imported from '../models/Artist,' creating a connection to the MongoDB database. Using the search method and the populate method for data enrichment, the findAllArtists method retrieves all artists along with the albums and tracks that go with them. If no artists are retrieved, the procedure throws an error and returns the array of artists. Based on the supplied artist data, the addArtist method generates a new artist instance, saves it to the database using the save function, and returns the newly formed artist. Error management is integrated to handle possible malfunctions that may arise during the creation or retrieval procedures. By exporting the artistService object, this code structure becomes more modular and manageable as other application components, such controllers, can use these methods for artist-related functions.

Follow:

Controllers: A followController object, defined in the 'FollowController.js' file, is in charge of responding to HTTP requests pertaining to user and artist actions that are followed in a Node.js application. The followService module, imported from '../Services/Follow', is what the controller uses to carry out the underlying business logic. In order to create a user-to-user follow connection, the followUser function processes a POST request. It does this by extracting the follower and user-to-follow IDs from the request body and parameters, respectively. In a similar vein, the followArtist method retrieves follower and artist-to-follow IDs from a POST request to create a follower-to-artist relationship. When an exception occurs, the methods return a 500 status code and error message. Both methods call the equivalent followService methods, which return a 200 status code and success message upon successful execution. The programme exports the 'followController' object for use in other places.

Model: The given code creates a MongoDB schema and model for managing the relationships between users and artists in a Node.js application that uses

Mongoose. Three fields are defined in the `FollowSchema}: 'follower,' which is the user who is starting the follow action and refers to the 'userlogin' model; 'followingUser,' which refers to the 'userlogin' model and indicates the user being followed; and 'followingArtist,' which refers to the 'Artist' model and indicates the artist being followed. Using Mongoose's `model` function, the resulting `FollowModel` is built and associated with the given schema and the 'Follow' name. This model serves as an interface between the application and the MongoDB collection called "follows," allowing the application to carry out CRUD operations on documents that reflect follow connections between users and artists.

Routes: The file 'FollowRoutes.js' sets up an Express.js router called 'followRouter,' which is used to manage HTTP requests related to user and artist actions that are followed in a Node.js application. This router creates two routes by importing the Express {Router` and the `followController` from their respective directories. The `followUser` method in the `followController` receives requests via the first route, a POST route at '/user,' which makes it easier for users to follow one another. The `followArtist` method from the `followController` is called by the second POST route at '/artist,' which controls the establishment of follower-to-artist relationships. For improved code organisation and maintainability, this modular design adheres to best practices in Express.js development by isolating route definitions from the controller logic that underlies them. After that, the 'followRouter' is exported so that it can be used in other areas of the application.

Services: The business logic for user and artist following actions in a Node.js application is contained in the 'followService' module. It connects to the MongoDB database to manage follow relationships by using the 'FollowModel' from '../models/Follow'. The 'followUser' method, which checks for pre-existing follow relationships before establishing a new one, makes sure that a user is not followed more than once. In a similar vein, the validation of artist follow relationships is carried out by the 'followArtist' approach. If a follow relationship is found, both approaches raise errors and display helpful messages. This method avoids redundant follow entries and guarantees data integrity. The module uses asynchronous actions, exports the 'followService' object for application integration, and handles errors using try-catch blocks.

Lyrics:

Controllers: A Node.js application's `lyricsController` object, defined in the 'LyricsController.js' file, is in charge of responding to HTTP requests pertaining to lyrics operations. This controller contains the business logic for obtaining, adding, updating, and removing lyrics linked to a certain song by utilising the services offered by the `lyricsService} module from '../Services/Lyrics'. The basic CRUD actions are represented by the four asynchronous methods: `getLyrics`, `addLyrics`, `updateLyrics`, and `deleteLyrics}. Every method takes pertinent data out of the request body and arguments, assigns the task to the {lyricsService}, and then returns the necessary status codes and messages. Every method's error handling makes sure that any unforeseen problems that arise during the lyrical operations result in a 500 Internal Server Error response along with a helpful error message.

Model: Using Mongoose, the code sample creates a MongoDB schema and model for lyrical data handling in a Node.js application. 'trackId,' a reference to the 'Track' model that establishes a relationship between the lyrics and the related music track, and 'lyrics,' a mandatory string field that represents the actual lyrical content, define the LyricsSchema. Mongoose's `model` function creates the resultant `LyricsModel} and associates it with the 'Lyrics' name and the provided schema. This model functions as an interface for CRUD operations on documents in the 'lyrics' MongoDB collection. After that, the 'LyricsModel' is exported to be used in other areas of the application, offering an organised and effective way to handle and work with lyrics-related data stored in the MongoDB database.

Routes: An Express.js router called "lyricsRouter" is configured by the "Lyrics.js" file to handle HTTP requests related to lyrics operations in a Node.js application. The router creates four separate routes for GET, POST, PUT, and DELETE requests by importing the Express {Router} and the {lyricsController} from their respective directories. To retrieve lyrics for a particular track, use the GET route, which is mapped to the root path ('/'). This route directs requests to the `getLyrics` method within the `lyricsController`. By using the corresponding methods from the `lyricsController`, the POST, PUT, and DELETE routes for the same path make it easier to add, update, and remove lyrics, accordingly. By separating route definitions from the underlying controller functionality and adhering to Express.js rules, this modular approach promotes maintainability and fosters clarity..

Services: In a Node.js application, the business logic for lyrics-related actions is included in the 'LyricsService.js' module. By utilising the 'LyricsModel' found in '../models/Lyrics,' the module defines four asynchronous methods: 'getLyrics,' 'addLyrics,' 'updateLyrics,' and 'deleteLyrics.' The 'getLyrics' function uses the 'findOne' method to retrieve lyrics for a given track; it returns null if the lyrics cannot be located. When a track's lyrics already exist, the 'addLyrics' method throws an error; if not, it builds a new 'LyricsModel' instance and saves it to the database. Comparably, the 'updateLyrics' function modifies the lyrics that already exist for a given track or raises an exception if none are found. The 'deleteLyrics' function eliminates the lyrics associated with a certain song or raises an error if the lyrics cannot be located. In the event of a failure, each method has error handling to deliver informative error messages. This module helps with the separation of concerns and encourages code reusability and maintainability in the application by exporting the 'lyricsService' object.

Playlist:

Controller: The 'playlistController' module in this code is in charge of responding to HTTP requests pertaining to playlists in a web application. It is dependent on the 'playlistService' module, which probably uses Mongoose or a similar tool to communicate with a database and encapsulates the business logic for playlist-related actions. 'getPlaylists' retrieves every playlist and sends it as a JSON response; 'addPlaylist' creates a new playlist using the data in the request body and returns a success message with the ID of the newly created playlist; 'DeletePlaylist' deletes a playlist by its ID and returns a success message; and 'editPlaylist' updates a playlist's information. All of these asynchronous functions are contained in the 'playlistController' object. Every function makes use of the 'playlistService' to carry out the required tasks and responds to any possible errors by providing the client with the relevant error messages and status codes. This modular design makes it easier to divide up the work and incorporates playlist-related features into the web application's general routing and management.

Models: For the purpose of describing playlists in a MongoDB database, this code defines a Mongoose schema. The Mongoose library's `Schema` and `model` components are used by it. Fields like name (a necessary string), description (a string), creator (referring to the required 'userlogin' model), and tracks (an array of ObjectIds referencing the 'Track' model) are all specified in the 'PlaylistSchema', which describes the structure of a playlist document. The schema lists restrictions like the need for a creator and name. The code then uses the \`model} function to create a Mongoose model called 'PlaylistModel' that is connected to the 'Playlist' collection in the MongoDB database. In order to incorporate the 'PlaylistModel' into the Mongoose ORM for the storing and retrieval of playlist-related data in the MongoDB database, the 'PlaylistModel' is finally exported for usage in other areas of the application.

Routes: This code configures an Express router to handle HTTP routes associated with a web application's playlists. It imports the 'playlistController' module, which has functions for controlling playlist-related actions, and uses the `Router` component from the 'express' library. The `Router()` function is used to initialise the 'playlistRouter'. The router has four established routes: a POST request to add a new playlist, a DELETE request to remove an existing playlist, a GET request to obtain all playlists, and a PUT request to modify or update an existing playlist. A comparable function from the 'playlistController' is linked to each route. Because of its modular design, which encourages clear code organisation and concern separation, playlist-related routes and logic can be included into the Express application's general structure. In the end, the setup 'playlistRouter' is exported so that it may be utilised in the primary application file.

Services: The 'playlistService' module, defined by this code, contains the business logic for playlist management in a web application. Using the Mongoose ORM and the 'PlaylistModel' imported from the '../models/Playlist' module, it communicates with a MongoDB database. Four asynchronous functions are offered by the 'playlistService': 'findAllPlaylists' retrieves every playlist from the database, filling in the 'creator' and 'tracks' fields for a more thorough response; 'addPlaylist' creates a new playlist using the data supplied in the 'playlistInfo' parameter; 'DeletePlaylist' removes a playlist by its ID; and 'editPlaylist' modifies a playlist by its ID and returns the updated playlist. Every function manages possible mistakes and raises an error with an explanation if something goes wrong when interacting with the database. When exported, the 'playlistService' module facilitates the easy integration of playlist-related features into other application sections, encouraging modular and well-structured code.

Profile:

Controller: As part of a module, this code exports four asynchronous functions. The module's purpose is to manage web application processes pertaining to profiles. The first method, `getProfile`, sends a JSON response after retrieving a user's profile from the request body using the supplied `userId}. The `createProfile` method generates a new user profile by retrieving pertinent data (such as the userId, bio, and profile image) from the request body. It then assigns the creation task to the `profileService} and returns the freshly created profile along with a success message. `likeTrack` and `unlikeTrack`, the third and fourth functions, manage like and unliking actions for a particular track that is identified by `trackId}. They use the `profileService` to update the user's profile, and they return the updated profile in response. The code responds with an error message and a 500 Internal Server Error status in the event that an error occurs during any operation.

Model: A Mongoose schema for a user profile in a MongoDB database is defined by this code. The Mongoose library's `Schema` and `model` components are used by it. The user (referring the 'userlogin' model), bio, profileImage, likedTracks (each identified by their ObjectId and referencing the 'Track' model), and followers (referencing the 'userlogin' model) are all included in the `ProfileSchema}. The schema outlines some limitations, including the need for a user, the need for the user to be unique, and the provision of default values for the bio and profileImage fields. Then, using the `model} function and the schema, a Mongoose model called 'ProfileModel' is created, and it is exported to be utilised in other areas of the application. This structure makes it easier to connect with the MongoDB database, which enables the retrieval and manipulation of user profiles.

Routes: This code configures an Express router in a web application to handle HTTP routes related to profiles. In order to handle various profile operations, it imports the `Router` component from the 'express' library and the 'profileController' module. Next, the `Router()` function is used to initialise the `profileRouter`. The router is used to define four routes: two POST requests for like and unliking a track, a GET request to obtain a user's profile, and a POST request to create a new profile. A comparable function from the 'profileController' is linked to each route. In order to handle profile-related HTTP requests, the specified `profileRouter` is finally exported for usage in the main application file, enabling integration with the Express application as a whole. The application's routing and controller logic are organised cleanly and concerns are kept apart thanks to this modular design.

Services: This code defines a module whose export functions are in charge of utilising the Mongoose ORM to interface with a MongoDB database and carry out different tasks pertaining to user profiles. A user's profile can be retrieved using the `findProfileByUserId` function, a new user profile can be created with a specified bio and profile image using the `createProfile` function, a track can be added to the list of tracks that the user has liked in their profile using the `likeTrack` function, and a track can be removed from the list of liked tracks using the `unlikeTrack` function. Every function contains the necessary Mongoose queries and communicates with the database using {ProfileModel} from the '../models/Profile' module. The functions enable for better error management and logging in the application by throwing an error with a descriptive message in the event that any errors occur during these operations. The asynchronous nature of these functions is indicated by the use of the `async/await` pattern, ensuring proper handling of promises in JavaScript.

Track:

Controller: The controller module for managing HTTP requests pertaining to tracks in a web application is defined by this code. It depends on the `trackService` module, which probably uses Mongoose or a similar tool to communicate with a database and encapsulates the business logic for track-related actions. The four asynchronous functions that make up the `trackController` object are as follows: `getTracks` returns all tracks, `addTrack` builds a new track using the data in the request body, and `likeTrack` and `unlikeTrack` handle operations to like and unlike a particular track, respectively. Every method responds to the client with the relevant status codes and messages after carrying out the required activities using the {trackService`. If there are any problems with these operations, the controller notifies the client of the problem by sending an error message and a 500 Internal Server Error status. Lastly, the main application uses the `trackController` object that has been exported, which makes it easier to incorporate track-related features into the web site's general routing and handling.

Model: For the purpose of describing tracks in a MongoDB database, this code defines a Mongoose schema. The Mongoose library's `Schema` and `model` components are utilised by it. The structure of a track document is specified by the `TrackSchema}, which contains fields like the required strings for title and artist, the album (which references the 'useralbum' model), duration, genre, and an array of likes, all of which are identified by their ObjectId and reference the 'userlogin' model. Constraints like the mandatory nature of the artist fields and title are established by the schema. The code then uses the `model} function to construct a Mongoose model called 'TrackModel,' which is exported to be used in other areas of the application. The application can store and retrieve track information, including associations with albums and user likes, thanks to this structure's interaction with the MongoDB database.

Routes: In order to handle HTTP routes connected to tracks in a web application, this code configures an Express router. In order to handle various track activities, it imports the 'trackController' module and uses the `Router` component from the 'express' library. The `Router()` function is used to initialise the `trackRouter`. The router has four configured routes: two POST requests for like and unliking tracks, a GET request to obtain all tracks, and a POST request to add a new track. A comparable function from the 'trackController' is linked to each route. Ultimately, the setup `trackRouter} is exported so that it may be utilised in the primary application code. Because of its modular design, which encourages clear code organisation and concern separation, track-related routes and logic can be integrated into the Express application structure as a whole.

Services: The 'trackService' module, which is defined by this code, contains the business logic for handling tracks in an online application. It uses the Mongoose ORM to communicate with the MongoDB database by importing the 'TrackModel' from the '../models/Track' module. Four asynchronous functions are provided by the 'trackService': 'findAllTracks' retrieves all tracks from the database and fills in the 'album' field; 'addTrack' makes a new track based on the track information that is provided; 'likeTrack' lets users like a particular track, guaranteeing that likes are unique; and 'unlikeTrack' lets users dislike a track by taking their like out of the 'likes' array. Every function manages possible mistakes and raises an error with an explanation if something goes wrong when interacting with the database. The 'trackService' module in its whole is afterwards exported for utilisation in other sections of the programme, aiding in the division of responsibilities and permitting the incorporation of track-related features into the program's general structure.

Useralbum:

Controller: The 'albumController' module in this code is in charge of responding to HTTP requests pertaining to albums within a web application. It depends on the 'albumService' module, which probably uses Mongoose or a similar tool to communicate with a database and encapsulates the business logic for album-related actions. The asynchronous functions on the 'albumController' object are as follows: 'getAlbums' retrieves all albums and returns them as a JSON response, while 'addAlbum' creates a new album based on the data in the request body and returns a success message with the ID of the newly formed album. Every function makes use of the 'albumService' to carry out the required tasks and responds to any possible problems by providing the client with the relevant error messages and status codes. The web application's general routing and handling may be more easily integrated with album-related capabilities thanks to the modular structure.

Model: A Mongoose schema for storing albums in a MongoDB database is defined by this code. The Mongoose library's `Schema` and `model` components are utilised by it. The 'AlbumSchema' lists the fields that make up an album document, including the title, artist, release date, and genre (all needed strings). Additional characteristics may be introduced in the schema based on the particular needs. The `model` function is used to construct a Mongoose model called 'AlbumModel' after the schema description. This model is connected to the MongoDB database's 'useralbum' collection. In order to incorporate the 'AlbumModel' into the Mongoose ORM for the storage and retrieval of album-related data in the MongoDB database, the model is finally exported for usage in other areas of the application.

Routes: This code configures an Express router to handle HTTP routes associated with a web application's albums. In order to manage album-related actions, it imports the 'albumController' module and uses the `Router` component from the 'express' library. The `Router()` function is used to initialise the 'albumRouter'. The router has two configured routes: a POST request to add a new album and a GET request to fetch all albums. A comparable function from the 'albumController' is linked to each route. The configured 'albumRouter' is then exported so that it can be used in the main application code. Because of its modular design, which promotes clear code organisation and concern separation, album-related routes and logic may be integrated into the Express application structure as a whole.

Services: The 'albumService' module, which is defined by this code, contains the business logic for handling albums in a web application. It uses the Mongoose ORM to communicate with a MongoDB database using the 'AlbumModel' imported from the '../models/useralbum' module. Two asynchronous functions are offered by the 'albumService': 'findAllAlbums' fetches every album from the database, and 'addAlbum' creates a new album based on the album information supplied. Every function manages possible mistakes and raises an error with an explanation if something goes wrong when interacting with the database. After that, the complete 'albumService' module is exported so that it may be used in other areas of the application. This helps to keep issues apart and makes it possible to incorporate album-related features into the application's overall architecture.

Userlogin:

ControllerThis code defines the 'userController' module, which is in charge of responding to user-related HTTP requests in a web application. It depends on the 'userService' module, which probably uses Mongoose or a similar tool to connect with a database and encapsulates the business logic for operations related to users. Two asynchronous functions are included in the 'userController' object: 'getUsers' retrieves all users and delivers them as a JSON response, while 'addUser' adds a new user based on the data in the request body and returns a success message with the ID of the newly formed user. Every function makes use of the 'userService' to carry out the required tasks and responds to possible errors by providing the client with the relevant error messages and status codes.

Models: For the purpose of representing people in a MongoDB database, this code defines a Mongoose schema. The Mongoose library's `Schema` and `model` components are used by it. The structure of a user document is specified by the 'UserSchema,' which has fields like the mandatory strings password and username (the latter having a unique constraint), as well as a boolean flag called 'banned', which is set to false by default. Additional features may be introduced in the schema based on particular needs. The code defines the schema first, then uses the \`model} function to create a Mongoose model called 'UserModel' that is connected to the 'userlogin' collection in the MongoDB database. In the end, the 'UserModel' is exported so that it can be utilised in other areas of the programme and coupled with the Mongoose ORM for the purpose of storing and retrieving user-related data from the MongoDB database.

Routes: The Express router defined by this code manages HTTP routes pertaining to users within a web application. In order to manage user-related tasks, it imports the 'userController' module and uses the `Router` component from the 'express' library. The `Router()` function is used to initialise the 'userRouter'. The router has two configured routes: a POST request to add a new user and a GET request to fetch all users. Nevertheless, a line is out of order: the POST route definition comes before the export statement module.exports = userRouter;. The export statement should come after both routes are defined in the proper order. Because to its modular design, which encourages clear code organisation and concern separation, user-related routes and logic can be included into the overall Eps.

Services: The 'userService' module, defined by this code, contains the business logic for handling users in an online application. It makes use of the 'bcrypt' package to hash passwords and Mongoose to communicate with a MongoDB database via the 'UserModel' from '../models/userlogin'. Three asynchronous functions are available in the module: 'findAllUsers' gets all users from the database; 'addUser' makes a new user with a hashed password; and 'getUserById' gets a user by ID. In addition to these tests, the 'addUser' method additionally checks to see whether the user is banned before permitting registration and if another user with the same email address already exists. Should any problems occur during these procedures, the functions will throw errors along with informative messages. Exporting the 'userService' module makes it simple to include these user-related features into other areas of the programme, encouraging modular and well-structured code.

- Some Problems faced while implementing:

- We encountered a problem while trying to use the external apis. The Apis that we were planning to use were with money and they used axios instead of ajax.

- Encountered a problem while trying to signup the users in our front end, giving us error 304 which had to do with the cors stopping the data from getting the data from the backend.

- Didn’t know how to link the frontend with the backend.

Abdelrahman 201908 Part ----

Further Analysis of the Backend Code for Spotify Clone

3.1. User Model and Schema

The User model defines the core information associated with each user. The schema includes:

username: Unique string identifier for the user.

email: Unique email address for user identification and communication.

password: Hashed user password for secure storage and authentication.

subscription: String value indicating the user's current subscription level (e.g., "free", "premium").

createdAt: Timestamp of user account creation.

This model provides a basic structure for user management and could be extended to include additional information, such as preferences, playlists, and listening history.

3.2. Authentication and JWT Implementation

The application utilizes JSON Web Tokens (JWT) for user authentication. This approach offers several advantages:

Secure: JWTs are signed and encrypted, making them tamper-proof and protecting against unauthorized access.

Stateless: Servers don't need to store session information, reducing reliance on storage resources.

Scalable:JWTs allow for efficient authentication across multiple servers and applications.

The authentication.js module handles JWT generation and verification. However, the provided code snippets don't explicitly show the implementation details of user login and token issuance. Further analysis would require examining the missing code to fully understand the authentication flow.

3.3. Subscription Management

The subscription.js module manages user subscriptions. It defines functions for:

cgetSubscription: Retrieves a user's current subscription information.

csetSubscription:Updates a user's subscription to a specified plan.

These functions utilize the subscription.js service, which likely interacts with a database or payment gateway to manage subscriptions. However, the provided code snippets do not reveal the specific implementation details of this service.

4.1. Additional Functionalities

Content Management: The application needs to handle music content, including storing song metadata, managing playlists, and providing streaming capabilities.

Search and Recommendation: Implementing search functionality and personalized recommendations would enhance user experience.

Integration with Music Providers: Integrating with external music providers would expand the available content library.

5.1. Security Considerations

Password Hashing: Utilizing strong hashing algorithms like bcrypt for password storage is crucial for user security.

API Security: Implement access control mechanisms like API keys or OAuth to restrict unauthorized access to backend resources.

Data Validation and Sanitization: Validate all user input and sanitize data to prevent injection attacks and other vulnerabilities.

6.1. Performance Optimization

Caching: Implement caching mechanisms to reduce database load and improve application responsiveness.

Database Optimization: Ensure database indexes are properly set up for efficient data retrieval.

Load Balancing: For large-scale deployments, consider load balancing to distribute traffic across multiple servers.