SONG DATA VISUALIZER

ROLLNO:UCE2023530:DISHA JANVE
UCE2023531:ADITI JOSHI
UCE2023532:JUI KULKARNI
UCE2023546:ANJALI PATEL

Introduction

In today's digital age, music streaming platforms have transformed how we discover, manage, and enjoy music. This project, *Song Data Visualiser*, aims to provide a simplified yet robust framework to manage and visualize song data efficiently. By leveraging key data structures, this application organizes user information, manages playlists, and offers personalized song recommendations.

The motivation behind creating *Song Data Visualiser* stems from the need for a structured way to handle extensive music data while providing users with a seamless experience in managing their playlists and listening history. Our project goes beyond basic playlist functionality by integrating personalized recommendations, recent song tracking, and a collaborative playlist feature (Party Mode) that prioritizes premium users.

Song Data Visualiser is designed to:

1. **Allow users to manage their playlists** through options to add, delete, or shuffle songs.
2. **Provide song recommendations** based on the genre of the last played song.
3. **Track the most recent songs** a user has listened to.
4. **Support a collaborative Party Mode**, where multiple users can contribute to a playlist, with priority given to premium members.

To achieve these features, we implemented a variety of data structures:

- **Arrays**: Used to store user data, such as usernames and account details.

- **Linked Lists**: Applied for playlists, enabling flexible song addition and removal.
- **Stacks**: Used for tracking the recent playing song and listening history, allowing Last-In-First-Out (LIFO) access.
- **Graphs**: Utilized for recommending songs by representing genres and linking songs that share similar characteristics.
- **HashMap**: Employed to map each song to its specific details, allowing for fast retrieval and unique identification of songs.

Key functions implemented in this project include:

1. **Playlist Management**: Users can add, delete, and shuffle songs within their playlists, enhancing customization and control over their listening experience.
2. **Song Recommendations**: The system recommends songs based on the genre of the last played song, enhancing user engagement by providing genre-specific suggestions.
3. **Recent Song Tracking**: By using a stack, the application keeps track of the most recently played songs, allowing users to revisit previously listened tracks effortlessly.
4. **Party Mode**: This collaborative feature allows multiple users to add songs to a shared queue, with priority given to premium users. This ensures that premium users' song choices are played first, adding a level of exclusivity and engagement during group listening sessions.

Tools and Technologies

- **Programming Language**: Java
- **Development Environment**: Eclipse.
- **Data Structures**: Main data structures used, such as:
    - Stack (for listening history).
    - Doubly Circular Linked List (for playlist).
    - Priority Queue
- **Database**: Database 'minipro' with entries for song.

System Design

A] Song Class

The `Song` class in the DS_Song_Data_Visualisation package represents a song entity, capturing essential attributes that define each song, such as its title, duration, genre, artist, album, tempo, and mood. This class is designed to store and manage song information effectively and provides a way to represent the song in a readable format. The class encapsulates the attributes of a song, ensuring that each instance can provide detailed information and be easily integrated into playlist and recommendation features in the Song Data Visualiser project.

Key Attributes -

1. name (String): Stores the name or title of the song. This is a fundamental attribute for identifying the song.

2. duration(String):Represents the duration of the song in a string format (e.g., `"00:04:35"` for 4 minutes and 35 seconds). This could be useful for displaying or managing play length within the playlist.

3. genre (String): Specifies the genre of the song (e.g., "Pop", "Rock", etc.). The genre attribute is integral to the recommendation feature, as it enables genre-based song suggestions for the user.

4. artist (String): Stores the artist's name who performed the song. This attribute allows for categorizing or sorting songs based on artists.

5. album(String): Indicates the album the song belongs to, providing context about the collection in which the song was released.

6. tempo(int): Represents the tempo of the song, which can help in classifying songs as fast-paced, slow, etc. This attribute could also support recommendation or playlist sorting based on mood or tempo.

7. mood (String): Describes the mood of the song (e.g., "happy", "sad", "neutral"). This attribute can enhance the user experience by allowing mood-based song sorting or filtering.

Methods-

1. toString() (@Override):

The `toString` method provides a custom string representation of a song. It returns a string in the format `"SongName by ArtistName from album AlbumName [Genre]`. This format is user-friendly and suitable for displaying song information in a console or UI, making it easy for users to identify songs at a glance.

2. getGenre() (String):

This method returns the genre of the song. It is a simple accessor method, allowing other parts of the program (such as recommendation algorithms) to retrieve the genre for filtering or suggesting similar songs.

3. getName() (String):

This method returns the name of the song. Like ‘getGenre’, this accessor method enables external classes to retrieve the song's name, which can be useful for playlist management or display.

Usage-

The `Song` class is a fundamental component in the *Song Data Visualiser* project, enabling the creation of song objects that can be stored in playlists, used in recommendations, and managed by various data structures (like arrays, linked lists, and hash maps).

```java
package DSA_SONG_DATA_VISUALIZATION;
public class Song {
        String name;
        String duration;  // duration in seconds
        String genre;
        String artist;
        String album;
        int tempo;
        String mood;

        public Song(String name, String duration, String genre, String artist, String album, int tempo, String mood) {
            this.name = name;
            this.duration = duration;
            this.genre = genre;
            this.artist = artist;
            this.album = album;
            this.tempo=tempo;
            this.mood=mood;
        }

        @Override
        public String toString() {
            return name + " by " + artist + " from album " + album + " [" + genre + "]";
        }
        String getGenre() {
            return genre;
        }
        String getName() {
            return name;
        }
    }
```

B] User Class

The `User` class in the *DS_Song_Data_Visualisation* package is responsible for managing user accounts and their associated playlists and listening histories within the *Song Data Visualiser* project. This class holds an array of users, initializes user details with sample data, and provides methods for user retrieval and displaying playlists.

Key Attributes-

1. users (`UserDetails[]`):
   An array of `UserDetails` objects representing users of the application. The array is hardcoded to contain five users with predefined usernames, passwords, premium statuses, playlists, and listening histories. This setup allows the class to simulate user data and interactions without requiring database integration.

Constructor-

- User()
  The constructor initializes the `users` array with five `UserDetails` objects, each having a unique username and premium status. It also creates Song and `Playlist` objects, assigns playlists to users, and populates each user's `ListeningHistory` stack with sample songs. This approach provides a pre-defined set of users with personalized playlists and histories.
  Songs Added to User Playlists:
  - Each `Playlist` object is initialized and populated with Song objects.
  - Playlists are then assigned to corresponding users, allowing each user to have a unique playlist.
- Listening History:
  - Each user's `ListeningHistory` stack is populated with sample Song objects.
  - This stack-based history structure allows for the recent song access feature, with the most recently played songs appearing at the top of the stack.

Methods-

1. getUserByUsername() (`public UserDetails getUserByUsername(String username)`)
   This method searches for a user in the `users` array based on their username. It performs the following steps:
   - Iterates through each `UserDetails` object in the `users` array.
   - If a matching username is found, it returns the `UserDetails` object for that user.
   - If no match is found, it returns `null`.

2. Usage: This method is useful for retrieving a specific user's details based on their username, enabling user-specific operations such as playlist and history access.
3. displayUserPlaylist() (`public void displayUserPlaylist(String username)`)
   This method displays the playlist of a specified user based on their username. It performs the following steps:
   - Iterates through the `users` array to find a user with a matching username.
   - If a match is found, it calls the `displayPlaylist()` method of the user's `Playlist` object to print out the playlist songs.
4. Usage: This method is useful for visually presenting a user's playlist, allowing other parts of the program to display or manipulate the playlist content.

UserDetails Class

The `UserDetails` class is a helper class nested within the `User` class. It stores individual user account details, such as username, password, premium status, playlist, and listening history. This class provides a foundational structure for defining user-related attributes and operations.

Key Attributes-

1. username (`String`):
   The unique username of the user.
2. password (`String`):
   The password associated with the user account. Currently, this is stored as plain text, but in a production setting, it would be securely hashed and encrypted.
3. Premium (`boolean`):
   Indicates whether the user has a premium account. Premium users receive certain privileges in features such as collaborative queue prioritization.
4. playlist (`Playlist`):
   A `Playlist` object representing the user's personal playlist. This playlist stores the songs added by the user.
5. ListeningHistory (`Stack<Song>`):
   A stack that stores the user's listening history, with the most recent songs at the top. This structure allows for efficient retrieval of recently played songs, which can be useful for features like recently played or repeat recommendations.

Methods-

1. isPremium() (`public boolean isPremium()`)
   Returns the premium status of the user. Premium users may have prioritized access to certain features, such as song priority in a collaborative playlist.

2. getListeningHistory() (`public Stack<Song> getListeningHistory()`)
   Returns the `ListeningHistory` stack for the user. This stack provides an ordered history of the songs that the user has recently played.
3. displayListeningHistory() (`public void displayListeningHistory()`)
   This method displays the songs in the user's listening history. It performs the following steps:
   - Checks if the `ListeningHistory` stack is empty. If empty, it outputs a message indicating no listening history.
   - If not empty, it prints each song in the stack, displaying the song name and artist.
4. Usage: This method is useful for presenting the user's listening history, allowing the application to provide a summary of recently played songs or support repeat-play functionality.

Usage-

The `User` and `UserDetails` classes play an essential role in managing user data within the *Song Data Visualiser* project. These classes facilitate user-related functionalities, such as:

- Storing and retrieving user information.
- Managing individual user playlists.
- Tracking listening history in a stack structure to enable "recently played" features.

The structure and organization of these classes allow for efficient user management and data handling, providing a solid foundation for implementing user-specific features in the *Song Data Visualiser* application.

```java
package DSA_SONG_DATA_VISUALIZATION;
import java.util.*;
public class User {
    UserDetails[] users;
    User() {
        users = new UserDetails[5];
        // Hardcoding the details of 5 users
        users[0] = new UserDetails("Alice123", "password", true);
        users[1] = new UserDetails("BobTheDog", "password", false);
        users[2] = new UserDetails("CATCF", "password", true);
        users[3] = new UserDetails("Dianasour", "password", false);
        users[4] = new UserDetails("ChristmasEVE", "password", true);
        Song song1 = new Song("Shape of You", "240", "Pop", "Ed Sheeran", "Divide", 96, "Happy");
        Song song2 = new Song("Blinding Lights", "200", "Synthwave", "The Weeknd", "After Hours", 88, "Energetic");
        Song song3 = new Song("Hallelujah", "300", "Classical", "Leonard Cohen", "Various Positions", 60, "Peaceful");
        Song song4 = new Song("Hotel California", "390", "Rock", "Eagles", "Hotel California", 75, "Nostalgic");
        Song song5 = new Song("Someone Like You", "285", "Pop", "Adele", "21", 70, "Sad");
        Song song6 = new Song("Closer", "250", "EDM", "Chainsmokers", "Collage", 120, "Romantic");
        Song song7 = new Song("Perfect", "263", "Pop", "Ed Sheeran", "Divide", 95, "Romantic");
        Song song8 = new Song("Memories", "195", "Pop", "Maroon 5", "Jordi", 80, "Melancholy");
        Playlist playlist1 = new Playlist();
        playlist1.addSongtoplaylist(song1);
        playlist1.addSongtoplaylist(song2);

        Playlist playlist2 = new Playlist();
        playlist2.addSongtoplaylist(song3);
        playlist2.addSongtoplaylist(song4);

        Playlist playlist3 = new Playlist();
        playlist3.addSongtoplaylist(song5);
        playlist3.addSongtoplaylist(song6);

        Playlist playlist4 = new Playlist();
        playlist4.addSongtoplaylist(song7);
        playlist4.addSongtoplaylist(song8);

        Playlist playlist5 = new Playlist();
```

```
        Playlist playlist5 = new Playlist();
        playlist5.addSongtoplaylist(song1);
        playlist5.addSongtoplaylist(song5);
        users[0].playlist = playlist1;
        users[1].playlist = playlist2;
        users[2].playlist = playlist3;
        users[3].playlist = playlist4;
        users[4].playlist = playlist5;
        users[0].ListeningHistory.push(song2);
        users[0].ListeningHistory.push(song3);

        users[1].ListeningHistory.push(song4);
        users[1].ListeningHistory.push(song5);

        users[2].ListeningHistory.push(song6);
        users[2].ListeningHistory.push(song7);

        users[3].ListeningHistory.push(song8);
        users[3].ListeningHistory.push(song1);

        users[4].ListeningHistory.push(song5);
        users[4].ListeningHistory.push(song2);
    }

     public UserDetails getUserByUsername(String username) {
            for (UserDetails user : users) {
                if (user.username.equals(username)) {
                    return user;
                }
            }
            return null;
        }
     public void displayUserPlaylist(String username) {
         for (UserDetails user : users) {
                if (user.username.equals(username)) {
                   user.playlist.displayPlaylist();
                }
```

```
                user.playlist.displayPlaylist();
            }
        }
}}
class UserDetails{
    String username;
    String password;
    boolean Premium;
    Playlist playlist;
    Stack<Song> ListeningHistory=new Stack<>();
    public UserDetails() {}
    public UserDetails(String username, String password, boolean isPremium) {
        this.username = username;
        this.password=password;
        this.Premium = isPremium;
    }
    public boolean isPremium() {
        return this.Premium;
    }
    public Stack<Song> getListeningHistory() {
        return this.ListeningHistory;
    }
    public void displayListeningHistory() {
        if (ListeningHistory.isEmpty()) {
            System.out.println("No songs in listening history for user: " + username);
        } else {
            System.out.println("Listening history for user: " + username);
            for (Song song : ListeningHistory) {
                System.out.println("- " + song.name + " by " + song.artist);
            }
        }
    }
}
```

C] PlayList Class

The `Playlist` class in the *DS_Song_Data_Visualisation* package manages a circular doubly linked list (DCLL) structure to store and manipulate songs in a user's playlist. This design enables efficient playlist operations such as adding, deleting, displaying, shuffling songs, and generating song recommendations.

Key Attributes-

1. head (`Node`):
   The head of the doubly circular linked list, which points to the first song in the playlist. In the DCLL structure, the `head` node's `prev` pointer links to the last node, and the `next` pointer of the last node points back to the `head`, forming a circular structure.
2. size (`int`):
   Stores the current number of songs in the playlist, allowing easy tracking of the playlist's length.

Methods-

1. addSongtoplaylist() (`public void addSongtoplaylist(Song song)`)
   Adds a new song to the playlist. This method creates a `Node` for the song and places it at the end of the circular linked list, maintaining the circular structure.
   - If the playlist is empty, the new node becomes the `head`.
   - If the playlist already has nodes, the new node is inserted between the `head` and the last node (`head.prev`).
   - The size of the playlist is incremented by 1.
2. deleteSongfromplaylist() (`public void deleteSongfromplaylist(String songName)`)
   Removes a song from the playlist by its name. It iterates through the nodes, searching for a node containing the specified song name.
   - If the song is found, it updates the `prev` and `next` pointers of adjacent nodes to remove the node from the list.
   - If the node being removed is the `head`, the `head` is reassigned to the next node.
   - The playlist size is decremented by 1.
3. Usage: This method is essential for playlist management, enabling users to remove unwanted songs from their playlist.
4. shuffleplaylist() (`public void shuffleplaylist()`)
   Shuffles the songs in the playlist randomly by copying the DCLL structure into an `ArrayList`, performing a Fisher-Yates shuffle, and then reassigning songs in the nodes according to the shuffled list.
   Usage: Useful for providing a varied listening experience, allowing the user to shuffle their playlist's order.
5. displayPlaylist() (`public void displayPlaylist()`)
   Displays all songs in the playlist in the order they appear in the circular doubly linked list.
   - If the playlist is empty, it outputs a message indicating no songs are in the playlist.
   - Otherwise, it iterates through each node, printing out each song.
6. Usage: This method is essential for visually presenting the playlist, allowing users to see all songs in their current order.
7. getSize() (`public void getSize()`)
   Prints the number of songs in the playlist.

Usage: Provides a quick summary of the playlist size, useful for feedback and display purposes.

8. recommendSongsBasedOnLast() (`public void recommendSongsBasedOnLast(HashMap<String, Song> allSongs, Map<String, List<String>> genreToSongs)`)
   Recommends songs based on the genre of the last song added to the playlist.
   - Determines the genre of the last song (`head.prev.song.genre`) in the playlist.
   - Retrieves a list of songs from the `genreToSongs` adjacency list that match this genre.
   - Filters out any songs already present in the playlist to avoid duplication.
   - Displays recommended songs that match the genre but are not already in the playlist.
9. Usage: This feature enhances user engagement by providing personalized recommendations, promoting music exploration within genres that the user enjoys.

Node Class

The `Node` class is an inner class within `Playlist` and represents individual nodes in the circular doubly linked list. Each node stores a `Song` and has pointers to the next and previous nodes in the list.

Key Attributes-

1. song (`Song`):
   Holds the `Song` object associated with this node.
2. next (`Node`):
   Points to the next node in the playlist. If this node is the last node, `next` points back to the `head`, maintaining the circular structure.
3. prev (`Node`):
   Points to the previous node in the playlist. If this node is the `head`, `prev` points to the last node, completing the circular structure.

Usage-

The `Playlist` class, with its circular doubly linked list structure, provides an efficient way to manage and interact with songs in a playlist. Key functionalities include:

- Adding and removing songs: The circular structure enables constant-time insertion and deletion, making it responsive for playlist management.

- Shuffling and displaying: These methods offer users control over their listening experience, allowing them to view and randomize their playlists as needed.
- Recommendations: By suggesting similar songs, the class enhances user experience by introducing new songs that align with the user's listening preferences.

The Node class plays a crucial role in forming the circular doubly linked list, supporting the playlist's circular navigation and efficient data handling. Together, these classes provide a comprehensive solution for playlist management within the Song Data Visualiser project.

```java
package DSA_SONG_DATA_VISUALIZATION;

import java.util.*;
class Node {
    Song song;
    Node next;
    Node prev;

    public Node(Song song) {
        this.song = song;
        this.next = this; // Points to itself (Circular)
        this.prev = this; // Points to itself (Circular)
    }
}
public class Playlist {
    Node head;
    int size;
    public Playlist() {
        head = null;
        size = 0;
    }

    // Add a song to the playlist
    public void addSongtoplaylist(Song song) {
        Node newNode = new Node(song);

        if (head == null) {
            head = newNode;
        } else {
            Node lastNode = head.prev;
            lastNode.next = newNode;
            newNode.prev = lastNode;
            newNode.next = head;
            head.prev = newNode;
        }
        size++;
    }
```

```
        do {
            songs.add(current.song);
            current = current.next;
        } while (current != head);

        Random random = new Random();
        for (int i = songs.size() - 1; i > 0; i--) {
            int j = random.nextInt(i + 1);
            Song temp = songs.get(i);
            songs.set(i, songs.get(j));
            songs.set(j, temp);
        }

        // Step 3: Rebuild the DCLL with shuffled order
        current = head;
        for (Song song : songs) {
            current.song = song;
            current = current.next;
        }
        displayPlaylist();
    }

    // Display all songs in the playlist
    public void displayPlaylist() {
        if (head == null) {
            System.out.println("The playlist is empty.");
            return;
        }

        Node current = head;
        getSize();
        do {
            System.out.println(current.song);
            current = current.next;
        } while (current != head);
    }
```

```
    // Get the size of the playlist
   public void getSize() {
       System.out.println(size+" Songs");
    }

// Inside the Playlist class
public void recommendSongsBasedOnLast(HashMap<String, Song> allSongs, Map<String, List<String>> genreToSongs) {
    // Step 1: Get the genre of the last inserted song
    String lastInsertedGenre = head.prev.song.genre;

    // Step 2: Fetch songs of the same genre from the adjacency list
    List<String> songsOfSameGenre = genreToSongs.getOrDefault(lastInsertedGenre, new ArrayList<>());
    // Step 3: Filter out songs already in the playlist
    List<Song> recommendations = new ArrayList<>();
    Node current = head;
    Set<String> playlistSongNames = new HashSet<>(); // To track songs in the playlist
    do {
        playlistSongNames.add(current.song.name.toUpperCase());
        current = current.next;
    } while (current != head);

    for (String songName : songsOfSameGenre) {
        if (!playlistSongNames.contains(songName.toUpperCase())) {
            recommendations.add(allSongs.get(songName.toUpperCase()));
        }
    }

    // Step 4: Return recommendations
    if (recommendations.isEmpty()) {
        System.out.println("No new recommendations available for the genre: " + lastInsertedGenre);
    } else {
        for (Song song : recommendations) {
            System.out.println(song);
        }
    }}
}
```

D] SongDatabase Class

The `SongDatabase` class in the *DS_Song_Data_Visualisation* package serves as a centralized data repository for storing, retrieving, and managing song data within the *Song Data Visualiser* project. This class connects to a MySQL database, retrieves song information, organizes it into data structures, and provides methods for accessing and organizing songs by genre. The class leverages `HashMap` and adjacency lists to provide efficient access and grouping of song data based on different attributes.

Key Attributes-

1. Songs (`static HashMap<String, Song>`):
   A static `HashMap` that stores all songs loaded from the database, where each song title (in uppercase) is used as the key. This structure allows for fast lookups and unique identification of songs by their titles.

Methods-

1. getSongs() (`public static HashMap<String, Song>`)
   This method establishes a connection to the MySQL database and loads all songs into the `Songs` map. It performs the following steps:
   - Connects to the database using JDBC with the specified URL, username, and password.
   - Calls `loadSongsFromDatabase(Connection conn)` to retrieve and store songs from the database into the `Songs` map.
2. Exceptions: Throws `ClassNotFoundException` and `SQLException` to handle any issues related to database connection and SQL operations.
3. getgenretosongs() (`public static Map<String, List<String>>`)
   This method creates an adjacency list that maps each genre to a list of song titles belonging to that genre. It calls `createGenreToSongsAdjList(HashMap<String, Song> songMap)` and returns a `Map` where the genre is the key and the associated value is a list of songs in that genre.
4. loadSongsFromDatabase() (`public static HashMap<String, Song>`)
   This private helper method loads all songs from the database into a `HashMap` called `songMap`. It performs the following steps:
   - Executes an SQL query to retrieve song details, including title, genre, duration, tempo, mood, artist, and album.
   - Iterates through the result set and constructs a `Song` object for each record.
   - Converts the song title to uppercase and uses it as the key in `songMap` to store the song uniquely.
5. SQL Query:
   The query retrieves data from the `track`, `album`, and `artist` tables, joining them based on `albumID` and `artistID` to get complete song information.
   Returns:
   A `HashMap<String, Song>` where the key is the uppercase title and the value is the `Song` object.
6. createGenreToSongsAdjList() (`public static Map<String, List<String>>`)
   This helper method takes a `HashMap` of songs and organizes them into an adjacency list by genre. It performs the following steps:
   - Iterates over each `Song` object in the `songMap`.
   - For each song, it retrieves the genre and title.
   - Initializes a list for each genre if it doesn't already exist in `genreToSongs`.
   - Adds the song title to the list associated with its genre.
7. Returns:
   A `Map<String, List<String>>` where each key is a genre, and the value is a list of song titles within that genre. This adjacency list is helpful for displaying or recommending songs by genre.

Usage-

The `SongDatabase` class plays a crucial role in managing the song data for the Song Data Visualiser project. By connecting to a MySQL database, it ensures that the song data is dynamically loaded and organized. This class provides access to:

- A complete list of songs with all attributes.
- An adjacency list by genre, which is useful for genre-based song recommendation and efficient data retrieval.

The structure and organization of `SongDatabase` make it easy to integrate into features that require quick access to song information, genre-based grouping, and playlist or recommendation functionalities. This design ensures that song data is readily accessible for various components in the project, enhancing the overall efficiency and usability of the application.

```java
package DSA_SONG_DATA_VISUALIZATION;
import java.util.*;
import java.sql.*;
public class SongDatabase {
    static HashMap<String,Song> Songs;
    public static HashMap<String,Song> getSongs()throws ClassNotFoundException, SQLException{
        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/minipro","root","root");
        Songs=loadSongsFromDatabase(con);
        /*System.out.println("Genre to Songs Adjacency List:");
        genreToSongs.forEach((genre, songs) ->
            System.out.println(genre + " -> " + songs)
        );*/
        return Songs;
    }
    public static Map<String,List<String>> getgenretosongs() {
        Map<String, List<String>> genreToSongs = createGenreToSongsAdjList(Songs);
        return genreToSongs;
    }
    public static HashMap<String, Song> loadSongsFromDatabase(Connection conn) {
        HashMap<String, Song> songMap = new HashMap<>();
        String query = "SELECT t.title, t.genre, t.duration, t.tempo, t.mood, a.name AS artist, al.title AS album "
                     + "FROM track t "
                     + "JOIN album al ON t.albumID = al.albumID "
                     + "JOIN artist a ON al.ArtistID = a.ArtistID";

        try (Statement stmt = conn.createStatement(); ResultSet rs = stmt.executeQuery(query)) {
            while (rs.next()) {
                // Retrieve values from the result set
                String title = rs.getString("title");
                String genre = rs.getString("genre");
                String duration = rs.getString("duration"); // assuming duration is in seconds
                int tempo = rs.getInt("tempo");
                String mood = rs.getString("mood");
                String artist = rs.getString("artist");
                String album = rs.getString("album");
```

```
                // Create a new Song object for each track
                Song song = new Song(title, duration, genre, artist, album, tempo, mood);

                // Convert song title to uppercase and use it as the key in the HashMap
                songMap.put(title.toUpperCase(), song); // Storing in HashMap with uppercase title as key
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return songMap; // Return the HashMap of songs
    }
    public static Map<String, List<String>> createGenreToSongsAdjList(HashMap<String, Song> songMap) {
        Map<String, List<String>> genreToSongs = new HashMap<>();

        // Iterate through the songMap
        for (Song song : songMap.values()) {
            String genre = song.getGenre(); // Get the genre of the song
            String title = song.getName(); // Get the title of the song

            // Add the song title to the genre's list
            genreToSongs.putIfAbsent(genre, new ArrayList<>()); // Initialize the list if not present
            genreToSongs.get(genre).add(title); // Add the song title to the list
        }

        return genreToSongs; // Return the adjacency list
    }
}
```

E] SongQueueItem Class

The `SongQueueItem` class in the DS_Song_Data_Visualisation package represents an item in a song queue. Each `SongQueueItem` associates a `Song` with a `UserDetails` object, allowing for prioritization of users based on their status (premium or regular). This prioritization can be utilized for playlist management, ensuring premium users get preferential treatment in queue processing.

Key Attributes-

1. song (Song):
   Stores the song associated with this queue item. This allows the queue to manage which song is being requested by a particular user.
2. user (`UserDetails`):
   Holds the `UserDetails` object, representing the user who requested the song. This attribute enables prioritization based on user-specific properties, such as premium membership status.

Methods-

1. compareTo() (`public int compareTo(SongQueueItem other)`)
   Implements the `Comparable` interface, enabling comparison between `SongQueueItem`

objects based on user priority. This method determines the order of items in the queue, with premium users prioritized over regular users.
Comparison Logic:

- Premium vs Regular: If the current `SongQueueItem` belongs to a premium user and the other item belongs to a regular user, the current item is given higher priority (`return -1`).
- Regular vs Premium: If the current item belongs to a regular user and the other item belongs to a premium user, the other item has higher priority (`return 1`).
- Same Priority: If both users are either premium or regular, they are considered of equal priority (`return 0`).

2. Usage: This method is essential for maintaining a priority queue structure where `SongQueueItem` objects are sorted based on user membership status. It ensures that premium users' requests are handled before those of regular users, enhancing the user experience by providing premium members with a benefit in the queue.

Usage and Significance-

The `SongQueueItem` class is designed to work within a priority queue for handling song requests in the Song Data Visualiser project. By leveraging the `Comparable` interface, this class enables priority-based queue management, where premium users are served before regular users. Key applications include:

- Queue Management: The `compareTo()` method allows `SongQueueItem` objects to be sorted by user priority in a priority queue, ensuring premium users' requests are processed first.
- Enhanced User Experience: This class enables a differentiated service for premium users, adding value for premium memberships.

The `SongQueueItem` class is a critical component for implementing a fair and responsive song queue system in the Song Data Visualiser project, where users' membership status directly influences their position in the queue. This design can enhance user satisfaction and promote premium membership by providing tangible benefits within the application.

```java
import java.util.*;
public class SongQueueItem implements Comparable<SongQueueItem> {
    Song song;
    UserDetails user;
    public SongQueueItem(Song song, UserDetails user) {
        this.song = song;
        this.user = user;
    }

    // Compare based on user priority
    @Override
    public int compareTo(SongQueueItem other) {
        if (this.user.isPremium() && !other.user.isPremium()) {
            return -1; // Premium users have higher priority
        } else if (!this.user.isPremium() && other.user.isPremium()) {
            return 1;
        } else {
            return 0; // Same priority for both regular and premium
        }
    }
}
```

F] PartyMode Class

The `PartyMode` class in the DS_Song_Data_Visualisation package enables a shared queue for music requests, where users can add songs to a playlist that prioritizes premium members. This class uses a priority queue to manage song requests efficiently, allowing premium users to have their requests played before those of regular users. `PartyMode` is designed for group settings or events where multiple users may request songs, with a system to prioritize requests and manage the song queue.

Key Attributes-

1. partyQueue (`PriorityQueue<SongQueueItem>`):
   A priority queue that stores `SongQueueItem` objects, which represent song requests from users. This queue orders songs based on user priority (premium users first), leveraging the `compareTo` method in `SongQueueItem`.

Methods-

1. addSongToQueue(Song song, UserDetails user)
   Adds a song to the `partyQueue` with a specified user. A new `SongQueueItem` is created to encapsulate the song and user information, and the item is added to the priority queue.
   Parameters:

    - `Song song`: The song to be added to the queue.
    - `UserDetails user`: The user who is adding the song to the queue.
2. Usage: This method is called to queue up a song for playback. It enables users to contribute songs to the playlist, with priority given to premium users.
   Output: Prints a confirmation message indicating that the user has successfully added a song to the queue.
3. playNextSong()
   Plays the next song in the queue based on priority. This method retrieves and removes the highest-priority `SongQueueItem` from the `partyQueue`, and displays information about the currently playing song and the user who requested it.
   Usage: This method is called to play the next song in the queue, prioritizing premium users. It simulates the process of moving to the next song in a party setting.
   Output: Prints the song name and the user who requested it. If the queue is empty, it prints a message indicating that there are no songs to play.
4. displayQueue()
   Displays the current song queue in priority order. This method temporarily copies the items in `partyQueue` to a list, sorts it, and prints the details of each song in the queue, including the song name and the user who requested it.
   Usage: This method is useful for viewing the current queue status, showing which songs are next in line and who requested them.
   Output: Displays the song names and associated users in order of priority. If the queue is empty, it prints a message indicating that the queue is empty.

Usage and Significance-

The `PartyMode` class is a central component for handling song requests in a shared playlist environment within the Song Data Visualiser project. This class provides an interactive experience for users in party settings, allowing multiple users to contribute to a playlist. Key applications include:

- Request Management: By prioritizing premium users in the queue, `PartyMode` provides added value to premium memberships, creating a more inclusive experience for all users while rewarding premium users.
- Playback Control: The `playNextSong()` method helps manage playback by moving through the queue in order of priority, ensuring a seamless listening experience.
- Queue Visibility: The `displayQueue()` method allows users to see the current lineup of songs, enhancing transparency in song selection and enabling users to anticipate when their requests will be played.

The `PartyMode` class adds an engaging, community-oriented feature to the Song Data Visualiser project, allowing for dynamic song selection while honoring premium memberships. This class can improve user satisfaction by facilitating fair and responsive queue management in a group environment.

```java
package DSA_SONG_DATA_VISUALIZATION;
import java.util.*;
public class PartyMode {
        PriorityQueue<SongQueueItem> partyQueue;

        public PartyMode() {
            // PriorityQueue with default comparator (using compareTo in SongQueueItem)
            this.partyQueue = new PriorityQueue<>();
        }

        // Method to add a song to the queue
        public void addSongToQueue(Song song, UserDetails user) {
            SongQueueItem item = new SongQueueItem(song, user);
            partyQueue.add(item);
            System.out.println(user.username + " added song: " + song.name + " to the queue.");
        }

        // Method to play the next song (with the highest priority)
        public void playNextSong() {
            if (!partyQueue.isEmpty()) {
                SongQueueItem item = partyQueue.poll(); // Retrieves and removes the first element
                System.out.println("Now playing: " + item.song.name + " by " + item.user.username);
            } else {
                System.out.println("No songs in the queue.");
            }
        }
        public void displayQueue() {
            if (partyQueue.isEmpty()) {
                System.out.println("The queue is empty.");
                return;
            }

            // Create a temporary list to sort and display the queue
            List<SongQueueItem> tempList = new ArrayList<>(partyQueue);
            tempList.sort(null); // Sort using the natural order (based on compareTo in SongQueueItem)

            System.out.println("Songs in the queue:");
            for (SongQueueItem item : tempList) {
                System.out.println("Song: " + item.song.name + ", Added by: " + item.user.username);}}}
```

G] Main Class

The `Main` class in the DS_Song_Data_Visualisation package acts as the central interface for the Song Data Visualiser application, providing a command-line menu for users. It enables two main operational modes: Remote Mode for personalized playlist management and Party Mode for shared music experiences with a priority-based song queue. This class utilizes various components from the application, such as `User`, `Playlist`, `Song`, `UserDetails`, `PartyMode`, and `SongQueueItem`, allowing users to perform tasks like searching for songs, adding songs to playlists or queues, and viewing or playing songs.

Key Attributes-

1. Scanner sc
    - Instance of `Scanner`, used to capture user inputs from the console.
2. String songName, SongName, username
    - Temporary string variables to store user inputs like song names and usernames.
3. Song foundSong
    - A `Song` object used to temporarily hold a song result when a user searches by name.
4. HashMap<String, Song> Songs
    - This HashMap stores the song data from `SongDatabase`. The song names serve as keys, and `Song` objects are the values, enabling quick lookups by song name.
5. Playlist playlist
    - An instance of the `Playlist` class, used to handle playlist-related operations in Remote Mode.
6. PartyMode partyMode
    - An instance of the `PartyMode` class, which manages the song queue for Party Mode, prioritizing songs requested by premium users.
7. User obj
    - An instance of the `User` class, providing access to user details and operations associated with user profiles.
8. UserDetails user
    - A `UserDetails` object representing the current user, used in the session to perform actions on behalf of that user.
9. Map<String, List<String>> genreToSongs
    - A map that holds genre-to-song mappings from `SongDatabase`, used to suggest songs based on the genre of songs added to the playlist.

Functional Overview

The `Main` class initiates the application, providing a menu interface that enables users to switch between Remote Mode and Party Mode based on their preference.

Remote Mode-

Remote Mode is designed for individual use, where users can view, modify, and interact with their personal playlist and listening history. It includes options to view playlists, search for songs, add or remove songs, and display listening history.

Workflow in Remote Mode

1. Prompting User for Username

- The application first prompts users to enter their username. It then retrieves the `UserDetails` object corresponding to this username using the `User` class's `getUserByUsername()` method.

2. Menu Options in Remote Mode
   - After logging in, users can choose from the following options:
   - 1. View Playlist
     - Calls `displayUserPlaylist(username)` in the `User` class to display the user's current playlist.
     - Lists each song in the playlist with details, such as the song name, artist, and other metadata.
   - 2. Search Songs
     - Prompts the user to enter a song name to search. The input is normalized (converted to uppercase) to match the stored song names in the `Songs` HashMap.
     - If the song is found, it displays the song's details and offers additional actions:
       - Play the Song
         - The selected song is played, and it is added to the user's listening history using `ListeningHistory.push()`.
       - Add the Song to Playlist
         - Adds the song to both the user's personal playlist (`user.playlist`) and the global playlist (`playlist`).
         - After adding the song, the application recommends similar songs based on the genre of the newly added song, using `recommendSongsBasedOnLast(Songs, genreToSongs)`.
       - Play Playlist in Shuffle Mode
         - Calls `shuffleplaylist()` in the `Playlist` class, which shuffles and plays all songs in the user's playlist in a random order.
       - View Listening History
         - Calls `displayListeningHistory()` on the user object, which displays a history of all songs the user has listened to.
   - 3. Remove a Song from Playlist
     - Prompts the user to enter the song name they want to remove from their playlist.
     - Calls `deleteSongfromplaylist(songname)` to remove the specified song from the playlist.
   - 4. Exit
     - Exits Remote Mode and returns the user to the main menu.

Example User Flow in Remote Mode

1. User enters their username.
2. User chooses to view their playlist.
3. User searches for a song by name. If found, they can play it, add it to the playlist, shuffle-play their playlist, or view their listening history.
4. User decides to remove a song from their playlist.

Party Mode-

Party Mode is intended for a shared environment, such as a social gathering, where multiple users can add songs to a single queue. The `PartyMode` class manages the queue using a priority-based system that gives premium users higher priority in the song queue.

Workflow in Party Mode

1. Prompting for Actions in Party Mode
    - The application displays a separate menu for Party Mode, offering the following actions:
2. Menu Options in Party Mode
    - 1. Search and Add a Song to Queue
        - Prompts the user to enter the song name they want to add.
        - If the song is found in the `Songs` database, the user is asked to enter their username to retrieve their `UserDetails`.
        - Adds the song to the `partyQueue` in the `PartyMode` class by creating a `SongQueueItem` object with the song and user details. The queue automatically orders songs based on user priority (premium users have higher priority).
    - 2. Play Next Song from Queue
        - Retrieves and plays the next song in the queue using `playNextSong()`. This method removes the song from the queue after playing it.
    - 3. View Queue
        - Calls `displayQueue()` to show the current list of songs in the queue. The queue displays songs in the order they will be played, highlighting the priority of premium users over regular users.
    - 4. Exit
        - Exits Party Mode and returns the user to the main menu.

Example User Flow in Party Mode

1. User searches for a song and adds it to the queue.
2. User views the queue to see which songs are up next.
3. User plays the next song in the queue.
4. User exits Party Mode to return to the main menu.

Summary-

The `Main` class effectively organizes and coordinates the functionalities offered by various classes in the application, providing a cohesive user experience in both Remote Mode and Party Mode. Below is a breakdown of the key functionalities provided by this class:

1. Song Searching and Retrieval
    - Users can search for songs by name, view song details, and add songs to playlists or the party queue.
2. Playlist Management
    - In Remote Mode, users can add, view, and remove songs from their playlist. They can also shuffle-play their playlist and view their listening history.
3. Priority Queue for Party Mode
    - Party Mode manages a song queue that prioritizes requests from premium users. Users can add songs to the queue, view the queue, and play songs in the order of priority.
4. Song Recommendation
    - After adding a song to the playlist, users receive recommendations for similar songs based on genre, enhancing their music experience.
5. Listening History Tracking
    - Each song a user listens to is recorded in their listening history, allowing them to revisit previously played songs.

Class Collaboration-

The `Main` class orchestrates the interactions between multiple classes, including:

- User and UserDetails: For user data management, such as fetching user details, playlists, and listening history.
- Playlist: For managing the playlist, including adding/removing songs, shuffling, and recommending songs.
- PartyMode and SongQueueItem: For managing the priority queue in Party Mode, ensuring premium users get priority in song playback.
- SongDatabase: For fetching song data and genre mappings, used in song searching and recommendation.

This modular approach allows the application to expand in the future by adding new features or enhancing existing ones without significant restructuring. The `Main` class serves as a controller, coordinating different components to provide a seamless user experience in a console-based environment.

```java
package DSA_SONG_DATA_VISUALIZATION;
import java.util.*;
import java.sql.*;
public class Main {
    public static void main(String[] args)throws ClassNotFoundException, SQLException {
        Scanner sc=new Scanner(System.in);
        String songName,SongName,username;
        Song foundSong;
        HashMap<String,Song> Songs=SongDatabase.getSongs();
        Playlist playlist=new Playlist();
        PartyMode partyMode = new PartyMode();
        User obj=new User();
        UserDetails user;
        //PartyMode partyMode = new PartyMode();
        Map<String,List<String>> genreToSongs=SongDatabase.getgenretosongs();
        int ch,ch2;
        System.out.println("Menu:");
        System.out.println("1. Remote Mode");
        System.out.println("2. Party Mode");
        System.out.println("Enter your choice");
        ch=sc.nextInt();
        sc.nextLine();
        switch(ch) {
        case 1:do {
            System.out.println("Enter your Username");
            username=sc.nextLine();
            user=obj.getUserByUsername(username);
            System.out.println("Menu:");
            System.out.println("1. View Playlist");
            System.out.println("2. Search Songs");
            System.out.println("3. Remove a song from Playlist");
            System.out.println("4. To Exit");
            ch2=sc.nextInt();sc.nextLine();
            switch(ch2) {
            case 1:
                obj.displayUserPlaylist(username);
                break;
            case 2:
                System.out.println("Enter the song name to be searched:");
                songName = sc.nextLine();
                SongName=songName.toUpperCase();
                if (Songs.containsKey(SongName)) {
```

```
                foundSong = Songs.get(SongName);
                System.out.println("Song found: " + foundSong);
                System.out.println("What would you like to do?");
                System.out.println("1. Play the song");
                System.out.println("2. Add the song to the playlist");
                System.out.println("3. Play Playlist in shuffle mode");
                System.out.println("4. View Listening History");
                int action = sc.nextInt();
                sc.nextLine(); // Consume the newline
                switch(action) {
                case 1:
                    System.out.println("Now playing: " + foundSong.name);
                    user.ListeningHistory.push(foundSong);
                    break;
                case 2:
                     user.playlist.addSongtoplaylist(foundSong);
                        playlist.addSongtoplaylist(foundSong);
                        System.out.println("Song added to your playlist: " + foundSong.name);
                        System.out.println();
                        System.out.println("Other Songs you might like:");
                        // Recommend songs based on genre
                        playlist.recommendSongsBasedOnLast(Songs, genreToSongs);
                        break;
                case 3:
                    user.playlist.shuffleplaylist();
                    break;
                case 4:
                    user.displayListeningHistory();
                }
            }
            break;
            case 3:
                System.out.println("Enter the song to be removed");
                String songname=sc.nextLine();
                user.playlist.deleteSongfromplaylist(songname);
                break;
            }
        }while(ch2!=3);
            break;
        case 2:
                int choice;
                do{
```

```
                break;
            }
        }while(ch2!=3);
            break;
        case 2:
                int choice;
                do{
                System.out.println("1. Search and Add a Song to Queue.");
                System.out.println("2. Play Next Song from Queue.");
                System.out.println("3. To View Queue.");
                System.out.println("4. To Exit.");
                System.out.println("Enter Choice");
                choice=sc.nextInt();
                sc.nextLine();
                switch(choice) {
                case 1:System.out.println("Enter the song name to be searched: ");
                songName = sc.nextLine();
                SongName=songName.toUpperCase();
                if (Songs.containsKey(SongName)) {
                    foundSong = Songs.get(SongName);
                    System.out.println("Enter your username: ");
                    username = sc.nextLine();
                    user=obj.getUserByUsername(username);
                    partyMode.addSongToQueue(foundSong, user);
                }
                else
                    System.out.println("Sorry! We couldn't find the song that you were lookig for.");
                break;
                case 2:
                    partyMode.playNextSong();
                    break;
                case 3:
                    partyMode.displayQueue();
                    break;
                }
            }while(choice!=4);

            break;
        }}
}
```

OUTPUT

Menu:
1. Remote Mode
2. Party Mode
Enter your choice
1
Enter your Username

Alice123
Menu:
1. View Playlist
2. Search Songs
3. Remove a song from Playlist
4. To Exit
1
2 Songs
Shape of You by Ed Sheeran from album Divide [Pop]
Blinding Lights by The Weeknd from album After Hours [Synthwave]
Enter your Username
Alice123
Menu:
1. View Playlist
2. Search Songs
3. Remove a song from Playlist
4. To Exit
2
Enter the song name to be searched:
sorry
Song found: Sorry by Justin Bieber from album Purpose [Pop]
What would you like to do?
1. Play the song
2. Add the song to the playlist
3. Play Playlist in shuffle mode
4. View Listening History
1
Now playing: Sorry
Enter your Username
Alice123
Menu:
1. View Playlist
2. Search Songs
3. Remove a song from Playlist
4. To Exit
3
Enter the song to be removed
Blinding Lights
Song removed from playlist: Blinding Lights

Menu:
1. Remote Mode
2. Party Mode
Enter your choice
1
Enter your Username
CATCF
Menu:
1. View Playlist

2. Search Songs
3. Remove a song from Playlist
4. To Exit
1
2 Songs
Someone Like You by Adele from album 21 [Pop]
Closer by Chainsmokers from album Collage [EDM]
Enter your Username
CATCF
Menu:
1. View Playlist
2. Search Songs
3. Remove a song from Playlist
4. To Exit
2
Enter the song name to be searched:
sorry
Song found: Sorry by Justin Bieber from album Purpose [Pop]
What would you like to do?
1. Play the song
2. Add the song to the playlist
3. Play Playlist in shuffle mode
4. View Listening History
2
Song added to your playlist: Sorry
Other Songs you might like:
What Do You Mean? by Justin Bieber from album Purpose [Pop]
Shake It Off by Taylor Swift from album 1989 [Pop]
How You Get the Girl by Taylor Swift from album 1989 [Pop]
One and Only by Adele from album 21 [Pop]
Rolling in the Deep by Adele from album 21 [Pop]
Unstoppable by Sia from album This is Acting [Pop]
Reaper by Sia from album This is Acting [Pop]
Alive by Sia from album This is Acting [Pop]
Bird Set Free by Sia from album This is Acting [Pop]
Desperado by Rihanna from album Anti [Pop]
Love Yourself by Justin Bieber from album Purpose [Pop]
New Man by Ed Sheeran from album Divide [Pop]
Kiss It Better by Rihanna from album Anti [Pop]
Style by Taylor Swift from album 1989 [Pop]
Set Fire to the Rain by Adele from album 21 [Pop]
Turning Tables by Adele from album 21 [Pop]
Company by Justin Bieber from album Purpose [Pop]
Muskurane by Arijit Singh from album Tum Hi Ho Hits [Pop]
This Love by Taylor Swift from album 1989 [Pop]
Close to You by Rihanna from album Anti [Pop]
Purpose by Justin Bieber from album Purpose [Pop]
Dressed in Black by Sia from album This is Acting [Pop]
The Feeling by Justin Bieber from album Purpose [Pop]

The Greatest by Sia from album This is Acting [Pop]
Space Between by Sia from album This is Acting [Pop]
Children by Justin Bieber from album Purpose [Pop]
Elastic Heart by Sia from album This is Acting [Pop]
Tum Hi Ho by Arijit Singh from album Tum Hi Ho Hits [Pop]
Blank Space by Taylor Swift from album 1989 [Pop]
Shape of You by Ed Sheeran from album Divide [Pop]
All You Had to Do Was Stay by Taylor Swift from album 1989 [Pop]
What Do I Know? by Ed Sheeran from album Divide [Pop]
Phir Le Aya Dil by Arijit Singh from album Tum Hi Ho Hits [Pop]
Blinding Lights by The Weeknd from album After Hours [Pop]
Too Late by The Weeknd from album After Hours [Pop]
Perfect by Ed Sheeran from album Divide [Pop]
Yeah, I Said It by Rihanna from album Anti [Pop]
I Found a Boy by Adele from album 21 [Pop]
Galway Girl by Ed Sheeran from album Divide [Pop]
Chikni Chameli by Shreya Ghoshal from album Shreyas Melodies [Pop]
Alone Again by The Weeknd from album After Hours [Pop]
Save Your Tears by The Weeknd from album After Hours [Pop]
Cheap Thrills by Sia from album This is Acting [Pop]
Dil Dhadakne Do by Neha Kakkar from album Neha?s Party Hits [Pop]

Menu:
1. Remote Mode
2. Party Mode
Enter your choice
1
Enter your Username
CATCF
Menu:
1. View Playlist
2. Search Songs
3. Remove a song from Playlist
4. To Exit
2
Enter the song name to be searched:
sorry
Song found: Sorry by Justin Bieber from album Purpose [Pop]
What would you like to do?
1. Play the song
2. Add the song to the playlist
3. Play Playlist in shuffle mode
4. View Listening History
3
2 Songs
Someone Like You by Adele from album 21 [Pop]
Closer by Chainsmokers from album Collage [EDM]

Menu:
1. Remote Mode
2. Party Mode
Enter your choice
1
Enter your Username
Alice123
Menu:
1. View Playlist
2. Search Songs
3. Remove a song from Playlist
4. To Exit
2
Enter the song name to be searched:
sorry
Song found: Sorry by Justin Bieber from album Purpose [Pop]
What would you like to do?
1. Play the song
2. Add the song to the playlist
3. Play Playlist in shuffle mode
4. View Listening History
4
Listening history for user: Alice123
- Blinding Lights by The Weeknd
- Hallelujah by Leonard Cohen

Menu:
1. Remote Mode
2. Party Mode
Enter your choice
2
1. Search and Add a Song to Queue.
2. Play Next Song from Queue.
3. To View Queue.
4. To Exit.
Enter Choice
1
Enter the song name to be searched:
Sorry
Enter your username:
BobTheDog
BobTheDog added song: Sorry to the queue.
1. Search and Add a Song to Queue.
2. Play Next Song from Queue.
3. To View Queue.
4. To Exit.
Enter Choice

1
Enter the song name to be searched:
Blinding Lights
Enter your username:
Alice123
Alice123 added song: Blinding Lights to the queue.
1. Search and Add a Song to Queue.
2. Play Next Song from Queue.
3. To View Queue.
4. To Exit.
Enter Choice
3
Songs in the queue:
Song: Blinding Lights, Added by: Alice123
Song: Sorry, Added by: BobTheDog
1. Search and Add a Song to Queue.
2. Play Next Song from Queue.
3. To View Queue.
4. To Exit.
Enter Choice
2
Now playing: Blinding Lights by Alice123
1. Search and Add a Song to Queue.
2. Play Next Song from Queue.
3. To View Queue.
4. To Exit.

CONCLUSION

The application successfully demonstrates a robust and interactive song management system, offering users two distinct modes: **Remote Mode** and **Party Mode**. Through features like personalized playlists, listening history, song recommendations, and a dynamic party queue system, the application provides a rich user experience. Its intuitive interface, efficient error handling, and seamless integration with a song database ensure reliability and user satisfaction.

Moreover, the use of modular and object-oriented programming principles enhances the scalability and maintainability of the application. Features like genre-based recommendations and shuffle mode make the system engaging, while the Party Mode caters to collaborative and social use cases.

In essence, this project highlights the potential of combining Java's strong foundation with interactive user interfaces to build innovative, user-centric applications. Future enhancements could include integrating APIs for live song streaming or introducing machine learning algorithms for more advanced song recommendations. This project stands as a comprehensive and impactful solution in the realm of music management systems.