



CS353 DATABASE SYSTEMS

2017-2018 SPRING SEMESTER

MUSICHOLICS

DESIGN REPORT

GROUP 17

Esra Nur AYZ

Ezgi ÇAKIR

Metehan KAYA

Miraç Vuslat BAŞARAN

Table of Contents

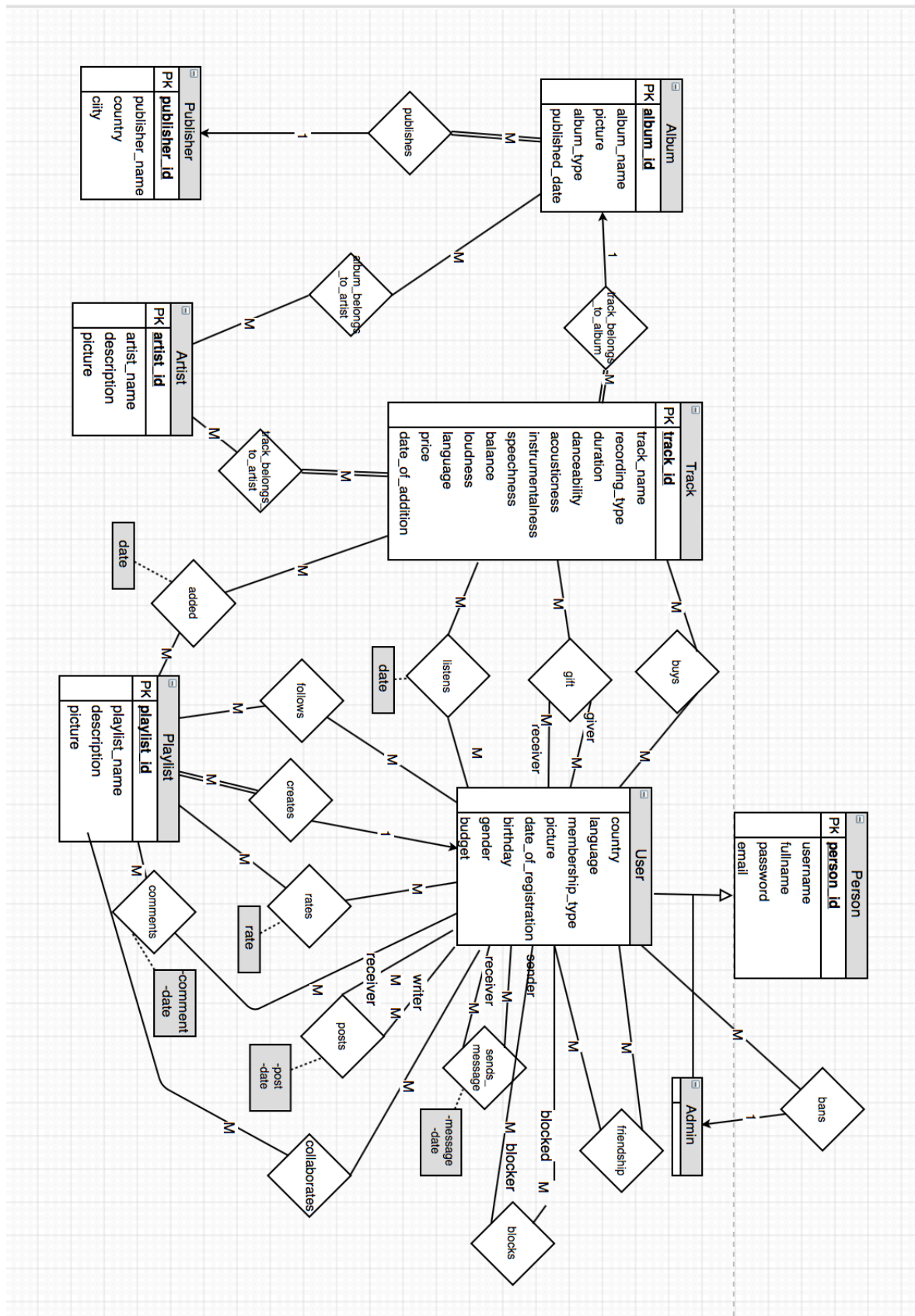
1. Revised E/R Model	6
2. Relation Schemas	8
2.1. Publisher	8
2.2. Album	9
2.3. Artist	10
2.4. Track	11
2.5. Album_Belongs_To_Artist	13
2.6. Track_Belongs_To_Artist	14
2.7. Playlist	15
2.8. Added	16
2.9. Person	17
2.10. User (extends Person)	18
2.11. Admin (extends Person)	19
2.12. Buys	20
2.13. Gift	21
2.14. Listens	22
2.15. Bans	23
2.16. Follows	24
2.17. Rates	25
2.18. Comments	26
2.19. Collaborates	27
2.20. Friendship	28
2.21. Blocks	29
2.22. Sends_Message	30
2.23. Posts	31
3. Functional Dependencies and Normalization of Tables	32
4. Functional Components	33
4.1. Use Cases and Scenarios	33
4.1.1. Admin	33
4.1.2. User	36
4.2. Algorithms	39
4.2.1. Searching Algorithms	39

4.2.2. Logical Requirements	39
4.3. Data Structures	40
5. User Interface Design and Corresponding SQL Statements	41
5.1. Home Page	41
5.2. Login	42
5.3. Register	43
5.4. Search Result Screen	44
5.4.1. Search Result Screen for Track	44
5.4.2. Search Result Screen for Playlist	45
5.4.3. Search Result Screen for Album	46
5.4.4. Search Result Screen for Artist	47
5.4.5. Search Result Screen for User	48
5.5. Change Personal Information	49
5.5.1. Change General Information	49
5.5.2. Change Password	51
5.6. Profile	52
5.6.1. Blocked Profile	52
5.6.2. Friend Profile	53
5.6.3. Nonfriend Profile	56
5.6.4. Own Profile	57
5.6.5. Complete Profile	58
5.7. Track	59
5.7.1. View Track	59
5.7.2. Access Track	61
5.7.3. Modify Track	63
5.8. Publisher	66
5.9. Artist	68
5.9.1. View Artist	68
5.9.2. Modify Artist	70
5.10. Album	72
5.10.1. View Album	72
5.10.2. Modify Album	74
5.11. Friends List	76
5.12. Playlist	78
5.12.1. View Playlists	78

5.12.2. View Own Playlist	80
5.12.3.View Another User's Playlist	83
5.13.Admin Panel	85
5.14.Messages	86
5.14.1. Message List	86
5.14.2.New Messages	87
5.15.Purchase	89
5.15.1.Purchase Track with Credit Card	89
5.15.2.Purchase Track with Budget	91
5.15.3.Purchase Premium with Credit Card	92
5.15.4.Purchase Premium with Budget	93
5.15.5.Purchase Premium with Budget	94
6. Advanced Data Components	95
6.1. Views	95
6.1.1. View a Friend's Profile	95
6.1.2. View a Profile of a User Not a Friend	95
6.1.3. Blocked Users	96
6.1.4. Message View	96
6.1.5. Album view	96
6.2. Stored Procedures	97
6.3. Reports	98
6.3.1. Total number of tracks in a playlist	98
6.3.2. Total number of followers of a playlist	98
6.3.3. Total number of comments of a playlist	98
6.3.4. Average rate of a playlist	98
6.3.5. Total number of playlist owned by a user	98
6.3.6. Total duration of a playlist	99
6.3.7. Total number of tracks of a album	99
6.3.8. Total number of user	99
6.3.9. Total number of tracks that is purchased	99
6.3.10. Total number of tracks that is listened by a user	99
6.3.11. Total number of collaborator of a playlist	99
6.3.12. Total number of friends of a user	100
6.3.13. Total number of chats of a user	100
6.4. Triggers	100

6.5. Constraints	101
7. Implementation Plan	102
8. Online Access	103

1. Revised E/R Model



We changed our E/R diagram of the database respect to the feedback we obtained from the teaching assistant. The changes are as follows:

- A super entity named “person” is created. “user” and “admin” entities are specification of that entity. Since username, fullname, password, and email is common between user and admin, those are passed to person entity. Since country, language, picture, date_of_registration, membership_type, birthday, gender, budget is just for user of the application, those remain in user entity.
- It is not removed “membership_type” attribute of “user” entity because we want to provide different limit of budget for different type of users.
- “send_messages” relation is added with message and date attributes. With this relation, users are able to send direct message each other.
- “date” attribute is added to “listens” in order to specify when a track is listened by the user.
- “gift” relation is turned to ternary relation which consist of 2 users and a track. One of users is giver of the gift, which is a track, and the other is receiver of the gift.
- “track_count”, “duration()”, and “num_followers” attributes are removed from “playlist” entity.
- “track_count()” attribute is removed from “album” entity.
- “post” and “date” attributes are added to “posts” relation.
- “collaborates” relation is added. By this function, a creator of a playlist can select his/her friends as collaborator to modify the playlist.
- “date” attribute is added to “comments” relation to store when the comment is written.

2. Relation Schemas

2.1. Publisher

Relational Model:

Publisher(publisher_id, publisher_name, country, city)

Functional Dependencies:

publisher_id \rightarrow publisher_name, country, city

publisher_name \rightarrow publisher_id, country, city

Candidate Keys:

{{publisher_id}, {publisher_name}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Publisher (  
    publisher_id INT NOT NULL AUTO_INCREMENT,  
    publisher_name VARCHAR(45) NOT NULL,  
    country VARCHAR(45) NOT NULL,  
    city VARCHAR(45) NOT NULL,  
    PRIMARY KEY(publisher_id),  
    UNIQUE(publisher_name) )
```


2.2. Album

Relational Model:

Album(album_id, album_name, picture, album_type, published_date, publisher_id)

Foreign Key: publisher_id to Publisher.

Functional Dependencies:

album_id → album_name, picture, album_type, published_date, publisher_id

Candidate Keys:

{{album_id}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Album (  
    album_id INT NOT NULL AUTO_INCREMENT,  
    album_name VARCHAR(45) NOT NULL,  
    picture_path VARCHAR(1024),  
    album_type ENUM ('single', 'album'),  
    published_date DATE NOT NULL,  
    publisher_id INT NOT NULL,  
    PRIMARY KEY(album_id),  
    FOREIGN KEY(publisher_id) REFERENCES Publisher(publisher_id) )
```

2.3. Artist

Relational Model:

Album(artist_id, artist_name, description, picture)

Functional Dependencies:

artist_id \rightarrow artist_name, description, picture

Candidate Keys:

{{artist_id}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Artist (  
    artist_id INT NOT NULL AUTO_INCREMENT,  
    artist_name VARCHAR(45) NOT NULL,  
    description VARCHAR(2048),  
    picture_path VARCHAR(1024),  
    PRIMARY KEY(artist_id) )
```

2.4. Track

Relational Model:

Album(track_id, track_name, recording_type, duration, danceability, acousticness, instrumentalness, speechness, balance, loudness, language, price, date_of_addition, album_id)

Foreign Key: album_id to Album.

Functional Dependencies:

track_id \rightarrow track_name, recording_type, duration, danceability, acousticness, instrumentalness, speechness, balance, loudness, language, price, date_of_addition, album_id

Candidate Keys:

{{track_id}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Track (  
    track_id INT NOT NULL AUTO_INCREMENT,  
    track_name VARCHAR(45) NOT NULL,  
    recording_type ENUM ('live', 'studio'),  
    duration TIME NOT NULL,  
    danceability DOUBLE(64,63),  
    acousticness DOUBLE(64,63),  
    instrumentalness DOUBLE(64,63),  
    speechness DOUBLE(64,63),  
    balance DOUBLE(64,63),  
    loudness DOUBLE(64,63),  
    language VARCHAR(45),  
    price DOUBLE (64, 32) NOT NULL,  
    date_of_addition DATE NOT NULL,  
    album_id INT NOT NULL,  
    PRIMARY KEY(track_id),  
    FOREIGN KEY(album_id) REFERENCES Album(album_id) )
```

2.5. Album_Belongs_To_Artist

Relational Model:

Album_Belongs_To_Artist (artist_id, album_id)

Foreign Key: artist_id to Artist

album_id to Album

Functional Dependencies:

(artist_id, album_id) → (artist_id, album_id) (Trivial)

Candidate Keys:

{{artist_id, album_id}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Album_Belongs_To_Artist (  
    artist_id INT NOT NULL,  
    album_id INT NOT NULL,  
    PRIMARY KEY(artist_id, album_id),  
    FOREIGN KEY(artist_id) REFERENCES Artist,  
    FOREIGN KEY(album_id) REFERENCES Album)
```

2.6. Track_Belongs_To_Artist

Relational Model:

Track_Belongs_To_Artist (track_id, artist_id)

Foreign Key: track_id to Artist

artist_id to Artist

Functional Dependencies:

(track_id, artist_id) \twoheadrightarrow (track_id, artist_id) (Trivial)

Candidate Keys:

{{track_id, artist_id}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Track_Belongs_To_Artist (  
    track_id INT NOT NULL,  
    album_id INT NOT NULL,  
    PRIMARY KEY(track_id, artist_id),  
    FOREIGN KEY(track_id) REFERENCES Track,  
    FOREIGN KEY(artist_id) REFERENCES Artist)
```

2.7. Playlist

Relational Model:

Playlist(playlist_id, playlist_name, description, picture, creator_id)

Functional Dependencies:

playlist_id → playlist_name, description, picture, creator_id

Candidate Keys:

{{person_id}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Playlist (  
    playlist_id INT NOT NULL AUTO_INCREMENT,  
    playlist_name VARCHAR(45) NOT NULL,  
    description VARCHAR(2048),  
    picture_path VARCHAR(1024),  
    creator_id INT NOT NULL,  
    PRIMARY KEY(playlist_id),  
    FOREIGN KEY(creator_id) REFERENCES User(user_id))
```

2.8. Added

Relational Model:

Playlist(playlist_id, track_id, date)

Functional Dependencies:

(playlist_id, track_id, date) → (playlist_id, track_id, date) (Trivial)

Candidate Keys:

{{playlist_id, track_id, date}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Added (  
    playlist_id INT NOT NULL,  
    track_id INT NOT NULL,  
    date TIMESTAMP NOT NULL,  
    PRIMARY KEY(playlist_id, track_id, date),  
    FOREIGN KEY(track_id) REFERENCES Track,  
    FOREIGN KEY(playlist_id) REFERENCES Playlist)
```


2.9. Person

Relational Model:

Person(person_id, username, fullname, password , email)

Functional Dependencies:

person_id \rightarrow username, fullname, password , email

username \rightarrow person_id, fullname, password, email

email \rightarrow person_id, username, fullname, password

Candidate Keys:

{{person_id}, (username), (email)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Person (  
    person_id INT NOT NULL AUTO_INCREMENT,  
    username VARCHAR(45) NOT NULL,  
    fullname VARCHAR(45),  
    password VARCHAR(45) NOT NULL,  
    email VARCHAR(128) NOT NULL,  
    PRIMARY KEY(person_id),  
    UNIQUE(username),  
    UNIQUE(email) )
```

2.10. User (extends Person)

Relational Model:

User(user_id, country, language, picture , date_of_registration, birthday, gender, budget)

Foreign Key: user_id to Person(person_id).

Functional Dependencies:

user_id → country, language, picture , date_of_registration, birthday, gender, budget

Candidate Keys:

{{user_id}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS User (  
    user_id INT NOT NULL,  
    country VARCHAR(45),  
    language VARCHAR(45),  
    picture_path VARCHAR(1024),  
    date_of_registration DATE,  
    birthday DATE,  
    gender ENUM('M', 'F', 'NB'),  
    budget DOUBLE(64,32),  
    PRIMARY KEY(user_id),  
    FOREIGN KEY(user_id) REFERENCES Person(person_id) )
```

2.11. Admin (extends Person)

Relational Model:

Admin(admin_id)

Foreign Key: admin_id to Person(person_id).

Functional Dependencies:

admin_id \rightarrow admin_id (Trivial)

Candidate Keys:

{{ admin_id }}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Admin (  
    admin_id INT NOT NULL,  
    PRIMARY KEY(admin_id),  
    FOREIGN KEY(admin_id) REFERENCES Person(person_id) )
```

2.12. Buys

Relational Model:

Buys(user_id, track_id)

Foreign Key: user_id to User

track_id to Track.

Functional Dependencies:

(user_id, track_id) \rightarrow (user_id, track_id) (Trivial)

Candidate Keys:

{{ user_id, track_id }}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Buys (  
    user_id INT NOT NULL,  
    track_id INT NOT NULL,  
    PRIMARY KEY(user_id, track_id) ,  
    FOREIGN KEY(user_id) REFERENCES User,  
    FOREIGN KEY(track_id) REFERENCES Track)
```

2.13. Gift

Relational Model:

Gift(giver_id, receiver_id, track_id)

Foreign Key: giver_id to User(user_id).

receiver_id to User(user_id).

track_id to Track.

Functional Dependencies:

(giver_id, receiver_id, track_id) → (giver_id, receiver_id, track_id) (Trivial)

Candidate Keys:

{{ giver_id, receiver_id, track_id }}

Normal Form:

BCNF

Table Definition:

CREATE TABLE IF NOT EXISTS Gift (

giver_id INT NOT NULL,

receiver_id INT NOT NULL,

track_id INT NOT NULL,

PRIMARY KEY(giver_id, receiver_id, track_id) ,

FOREIGN KEY(giver_id) REFERENCES User(user_id),

FOREIGN KEY(receiver_id) REFERENCES User(user_id),

FOREIGN KEY(track_id) REFERENCES Track)

2.14. Listens

Relational Model:

Listens(user_id, track_id, date)

Foreign Key: user_id to User

track_id to Track.

Functional Dependencies:

(user_id, track_id, date) \rightarrow (user_id, track_id, date) (Trivial)

Candidate Keys:

{{ user_id, track_id, date }}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Listens (  
    user_id INT NOT NULL,  
    track_id INT NOT NULL,  
    date TIMESTAMP NOT NULL,  
    PRIMARY KEY(user_id, track_id, date),  
    FOREIGN KEY(user_id) REFERENCES User,  
    FOREIGN KEY(track_id) REFERENCES Track)
```

2.15. Bans

Relational Model:

Bans(user_id, admin_id)

Foreign Key: user_id to User.

admin_id to Admin.

Functional Dependencies:

user_id \rightarrow admin_id

Candidate Keys:

{{ user_id }}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Bans (  
    user_id INT NOT NULL,  
    admin_id INT NOT NULL,  
    PRIMARY KEY(user_id) ,  
    FOREIGN KEY(user_id) REFERENCES User,  
    FOREIGN KEY(admin_id) REFERENCES Admin)
```

2.16. Follows

Relational Model:

Follows(user_id, playlist_id)

Foreign Key: user_id to User.

playlist_id to Playlist.

Functional Dependencies:

(user_id, playlist_id) \rightarrow (user_id, playlist_id) (Trivial)

Candidate Keys:

{{ user_id, playlist_id }}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Follows (  
    user_id INT NOT NULL,  
    playlist_id INT NOT NULL,  
    PRIMARY KEY(user_id, playlist_id) ,  
    FOREIGN KEY(user_id) REFERENCES User,  
    FOREIGN KEY(playlist_id) REFERENCES Playlist)
```


2.17. Rates

Relational Model:

Rates(user_id, playlist_id, rate)

Foreign Key: user_id to User.

playlist_id to Playlist.

Functional Dependencies:

(user_id, playlist_id) \rightarrow rate

Candidate Keys:

{{ user_id, playlist_id }}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Rates (  
    user_id INT NOT NULL,  
    playlist_id INT NOT NULL,  
    rate INT NOT NULL,  
    PRIMARY KEY(user_id, playlist_id) ,  
    FOREIGN KEY(user_id) REFERENCES User,  
    FOREIGN KEY(playlist_id) REFERENCES Playlist)
```

2.18. Comments

Relational Model:

Comments(user_id, playlist_id, date, comment)

Foreign Key: user_id to User.

playlist_id to Playlist.

Functional Dependencies:

(user_id, playlist_id, date) → comment

Candidate Keys:

{{ user_id, playlist_id, date }}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Comments (  
    user_id INT NOT NULL,  
    playlist_id INT NOT NULL,  
    comment VARHAR(2048) NOT NULL,  
    date TIMESTAMP NOT NULL,  
    PRIMARY KEY(user_id, playlist_id, date) ,  
    FOREIGN KEY(user_id) REFERENCES User,  
    FOREIGN KEY(playlist_id) REFERENCES Playlist)
```

2.19. Collaborates

Relational Model:

Collaborates(user_id, playlist_id)

Foreign Key: user_id to User.

playlist_id to Playlist.

Functional Dependencies:

(user_id, playlist_id) \rightarrow (user_id, playlist_id) (Trivial)

Candidate Keys:

{{ user_id, playlist_id }}

Normal Form:

BCNF

Table Definition:

CREATE TABLE IF NOT EXISTS Follows (

user_id **INT NOT NULL**,

playlist_id **INT NOT NULL**,

PRIMARY KEY(user_id, playlist_id) ,

FOREIGN KEY(user_id) **REFERENCES** User,

FOREIGN KEY(playlist_id) **REFERENCES** Playlist)

2.20. Friendship

Relational Model:

Collaborates(user1_id, user2_id)

Foreign Key: user1_id to User(user_id).

user2_id to User(user_id).

Functional Dependencies:

(user1_id, user2_id) \rightarrow (user1_id, user2_id) (Trivial)

Candidate Keys:

{{user1_id, user2_id}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Friendship (  
    user1_id INT NOT NULL,  
    user2_id INT NOT NULL,  
    PRIMARY KEY(user1_id, user2_id) ,  
    FOREIGN KEY(user1_id) REFERENCES User(user_id),  
    FOREIGN KEY(user2_id) REFERENCES User(user_id))
```

2.21. Blocks

Relational Model:

Blocks(blocker_id, blocked_id)

Foreign Key: blocker_id to User(user_id).

blocked_id to User(user_id).

Functional Dependencies:

(blocker_id, blocked_id) \rightarrow (blocker_id, blocked_id) (Trivial)

Candidate Keys:

{{ blocker_id, blocked_id }}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Blocks (  
    blocker_id INT NOT NULL,  
    blocked_id INT NOT NULL,  
    PRIMARY KEY(blocker_id, blocked_id) ,  
    FOREIGN KEY(blocker_id) REFERENCES User(user_id),  
    FOREIGN KEY(blocked_id) REFERENCES User(user_id))
```

2.22. Sends_Message

Relational Model:

Sends_Message(sender_id, receiver_id, date, message)

Foreign Key: sender_id to User(user_id).

receiver_id to User(user_id).

Functional Dependencies:

(sender_id, receiver_id, date) → message

Candidate Keys:

{{ sender_id, receiver_id, date }}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Sends_Message (  
    sender_id INT NOT NULL,  
    receiver_id INT NOT NULL,  
    date TIMESTAMP NOT NULL,  
    message VARCHAR(2048) NOT NULL,  
    PRIMARY KEY(sender_id, receiver_id, date) ,  
    FOREIGN KEY(sender_id) REFERENCES User(user_id),  
    FOREIGN KEY(receiver_id) REFERENCES User(user_id))
```

2.23. Posts

Relational Model:

Posts(writer_id, receiver_id, date, post)

Foreign Key: writer_id to User(user_id).

receiver_id to User(user_id).

Functional Dependencies:

(writer_id, receiver_id, date) → post

Candidate Keys:

{{ writer_id, receiver_id, date }}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS Posts (  
    writer_id INT NOT NULL,  
    receiver_id INT NOT NULL,  
    date TIMESTAMP NOT NULL,  
    post VARCHAR(2048) NOT NULL,  
    PRIMARY KEY(writer_id, receiver_id, date) ,  
    FOREIGN KEY(writer_id) REFERENCES User(user_id),  
    FOREIGN KEY(receiver_id) REFERENCES User(user_id))
```

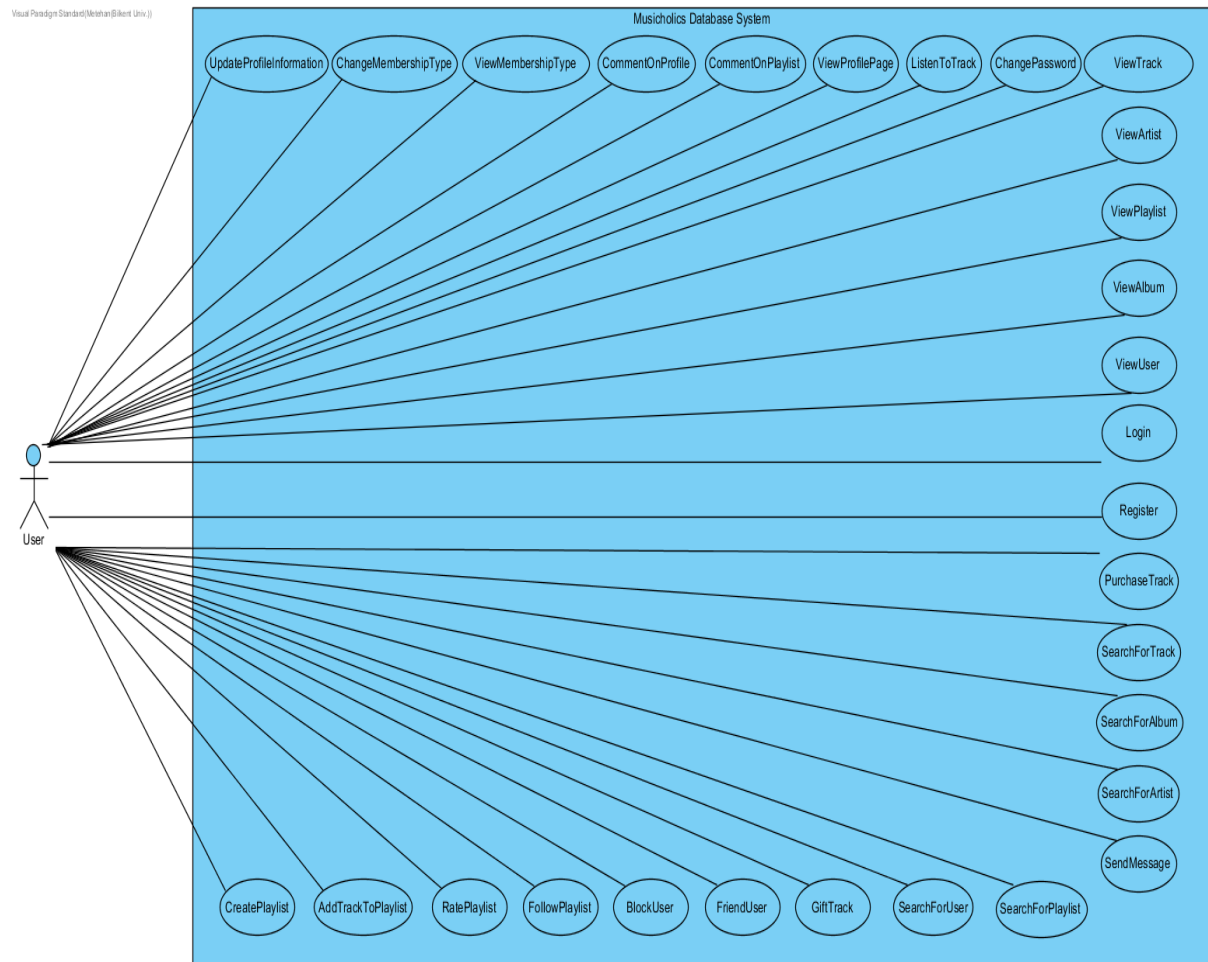
3. Functional Dependencies and Normalization of Tables

All relations are in Boyce-Code normal form and all the functional dependencies and normal forms are stated in Section 2 of the report.

- Login: The admin can login to the system with username and password.
- BanUser: The user can block a user if she posts inappropriate content, comments inappropriately, sends inappropriate direct message etc.
- AddTrack: The admin can add tracks to the system by including its price, recording type, and language.
- DeleteTrack: The admin can delete tracks from the system.
- ModifyTrack: The admin can modify information about tracks.
- AddAlbum: The admin can add albums to the system by including tracks.
- DeleteAlbum: The admin can delete albums from the system.
- ModifyAlbum: The admin can modify information about albums.
- AddArtist: The admin can add artists to the system by including her general information, tracks and albums.
- DeleteArtist: The admin can delete artists from the system.
- ModifyArtist: The admin can modify information about artists.
- AddPublisher: The admin can add publishers by specifying country and city.
- DeletePublisher: The admin can delete publishers.
- ModifyPublisher: The admin can modify information about publishers.
- ViewProfilePage: The admin can view her profile including information about her username, name, e-mail address, country, language, birth date, gender, playlists, wall.
- ViewTrack: The admin can view a track and access name, recording type, duration, language, price, date of addition and some features such as danceability, acousticness, instrumentalness, speechiness, balance, loudness of it. Also, albums including the track and artists who made the track will be visible.
- ViewPlaylist: The admin can view a playlist and access tracks, name, description, picture of it.
- ViewAlbum: The admin can view an album and access artist, publisher, name, picture, type, publish date, and tracks. Name, length, and price will be visible for each track.

- ViewArtist: The admin can view an artist and access name, picture and albums of her. Also, some description related to the artist will be visible. Name, type, and publish date will be visible for each album.
- ViewUser: The admin can view other users' profiles including information about her username, name, e-mail address, country, language, birth date, gender, playlists, wall.
- SearchForTrack: The admin can search for tracks from search menu by choosing track option.
- SearchForAlbum: The admin can search for albums from search menu by choosing album option.
- SearchForArtist: The admin can search for artists from search menu by choosing artist option.
- SearchForPlaylist: The admin can search for playlists from search menu by choosing playlist option.
- SearchForUser: The admin can search for users from search menu by choosing user option.

4.1.2. User



- **Login:** The user can login to the system with username and password.
- **Register:** The user can register to the system by specifying her name, surname, password, e-mail address, country, language, birth date, gender information and uploading her profile picture.
- **ListenToTrack:** The user can listen to tracks.
- **PurchaseTrack:** The user can purchase tracks by using her credit card.
- **SearchForTrack:** The user can search for tracks from search menu by choosing track option.
- **SearchForAlbum:** The user can search for albums from search menu by choosing album option.

- SearchForArtist: The user can search for artists from search menu by choosing artist option.
- SearchForPlaylist: The user can search for playlists from search menu by choosing playlist option.
- SearchForUser: The user can search for users from search menu by choosing user option.
- ChangePassword: The user can change her password. Entering old password is required to change the password.
- BlockUser: The user can block a user if she posts inappropriate content, comments inappropriately, sends inappropriate direct message etc.
- ViewProfilePage: The user can view her profile including information about her username, name, e-mail address, country, language, birth date, gender, playlists, wall.
- GiftTrack: The user can choose a track and send it as a gift to her friends.
- UpdateProfileInformation: The user can update her profile information by specifying name, e-mail address, country, language, birth date, gender, membership type, password.
- CommentOnProfile: The user can comment on other users' profiles. Comment will be visible on the wall.
- CommentOnPlaylist: The user can comment on other users' playlists. Comment will be visible on the wall.
- FriendUser: The user can friend other users from her profile page.
- FollowPlaylist: The user can follow other users' playlists. Playlists will be accessible from profile pages.
- RatePlaylist: The user can rate other users' playlists. Playlists will be accessible from profile pages.
- AddTrackToPlaylist: The user can add tracks to her playlists.
- CreatePlaylist: The user can create playlists by specifying the name.

- ViewTrack: The user can view a track and access name, recording type, duration, language, price, date of addition and some features such as danceability, acousticness, instrumentalness, speechiness, balance, loudness of it. Also, albums including the track and artists who made the track will be visible.
- ViewPlaylist: The user can view a playlist and access tracks, name, description, picture of it.
- ViewAlbum: The user can view an album and access artist, publisher, name, picture, type, publish date, and tracks. Name, length, and price will be visible for each track.
- ViewArtist: The user can view an artist and access name, picture and albums of her. Also, some description related to the artist will be visible. Name, type, and publish date will be visible for each album.
- ViewUser: The user can view other users' profiles including information about username, name, country, and age.
- SendMessage: The user can send messages to other users.
- ViewMembershipType: The user can view her membership type.
- ChangeMembershipType: The user can change her membership type.

4.2. Algorithms

4.2.1. Searching Algorithms

Users and admins can search for tracks, albums, playlists, artists, users. When a user or an admin search for a track, an album, a playlist, an artist or a user, checking all of the data and comparing keyword with all substrings of the string which is checked can consume a lot of time. If we assume that N is the length of the string and M is the length of keyword, then corresponding time complexity is $O(NM)$. If we also consider size of the data, searching can take a lot of time. Instead of doing a linear search and checking whether a given keyword matches with a substring of the word or not, an efficient search algorithm such as Knuth Morris Pratt (KMP) algorithm can be applied. This algorithm creates a table by using the given keyword, then uses it while searching the keyword in the string. Time complexity of this algorithm is $O(N+M)$. Alternative algorithms are Rabin-Karp and Boyer Moore algorithms which are not as sufficient as KMP algorithm.

4.2.2. Logical Requirements

Preventing logical errors are important for the database system, to have true information. The system should be designed in such a way that, it should not give wrong results to queries.

An example to boundary cases is that data should be compatible after insertions, deletions, and modifications. For instance, when an artist is removed from the database system, all related data of it should be deleted.

4.3. Data Structures

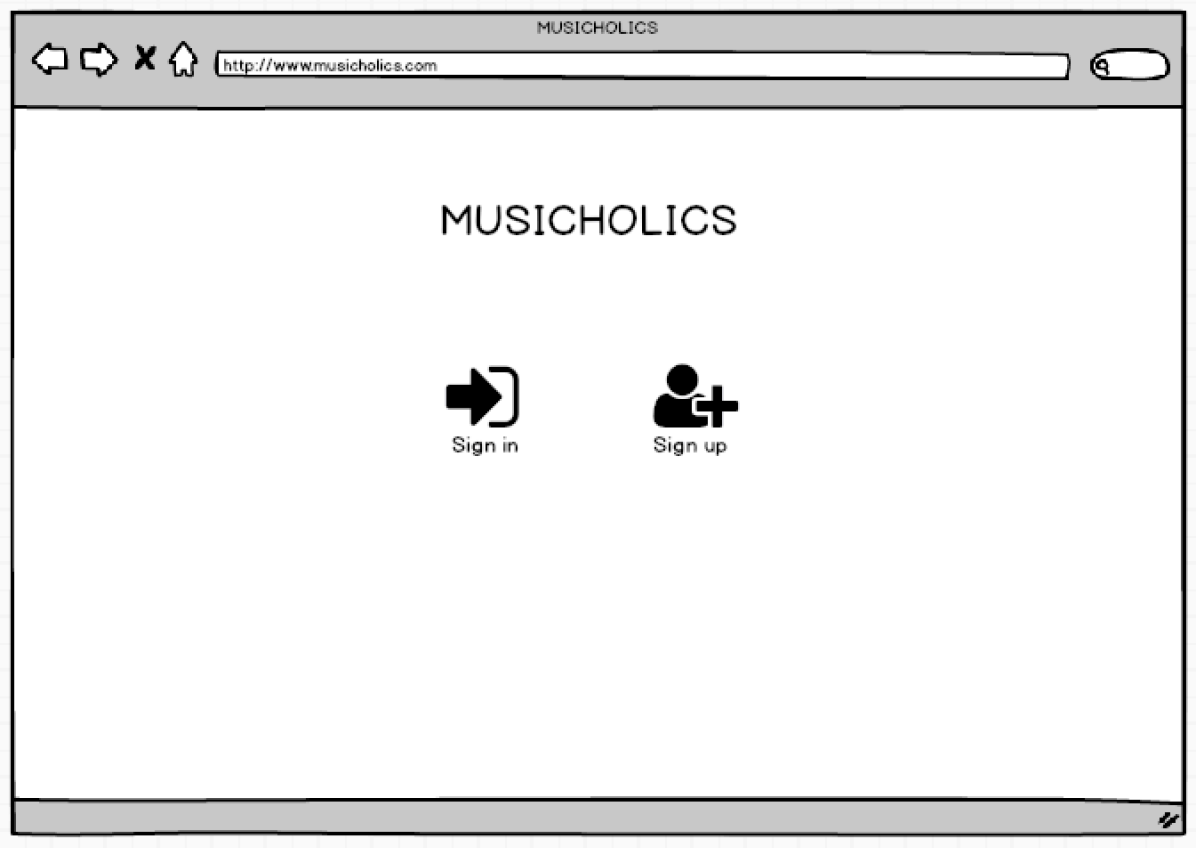
Numeric type, String type, Date type, and Time type of MySQL are used for the attribute domains.

Numeric types are required in order to store numeric data such as id for users, duration for tracks, track_count for playlists. For the numeric types, we used INT, and MEDIUMINT. MEDIUMINT is used for small integer values such as duration of a track.

String types are used in order to store any attributes composed of characters such as album_name for albums, country for publishers, description for artists, language for tracks, username for admins, email_address for users, etc.

5. User Interface Design and Corresponding SQL Statements

5.1. Home Page



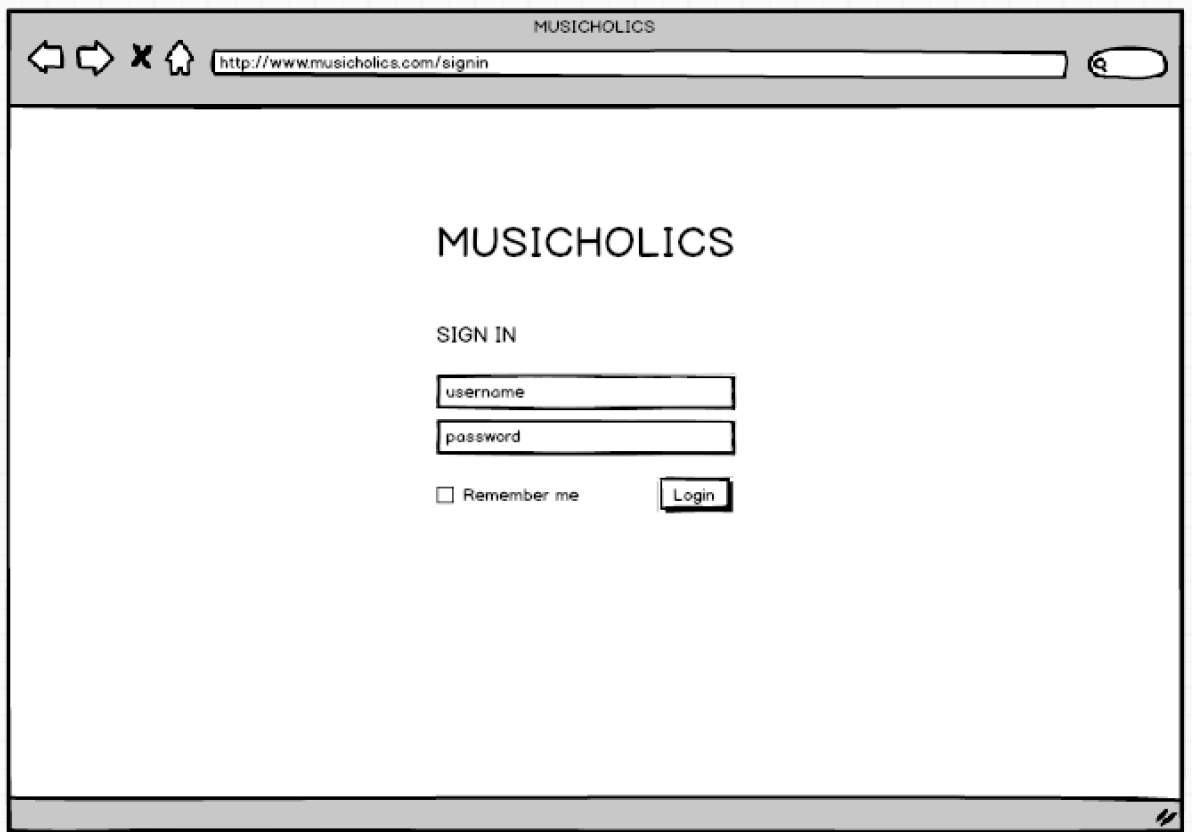
Inputs: No input

Process: The person (user or admin) chooses to enter or register.

SQL Statements

No SQL Statement is needed.

5.2. Login



The screenshot shows a web browser window with the title "MUSICHOLICS". The address bar contains the URL "http://www.musicholics.com/signin". The main content area displays the "MUSICHOLICS" logo at the top, followed by the text "SIGN IN". Below this, there are two input fields: "username" and "password". Under the "password" field, there is a checkbox labeled "Remember me" and a "Login" button. The browser window has standard navigation buttons (back, forward, stop, home) and a search icon in the top left corner.

Inputs: @username, @password

Process: The person (user or admin) enters his username and password to enter the Musicholics.

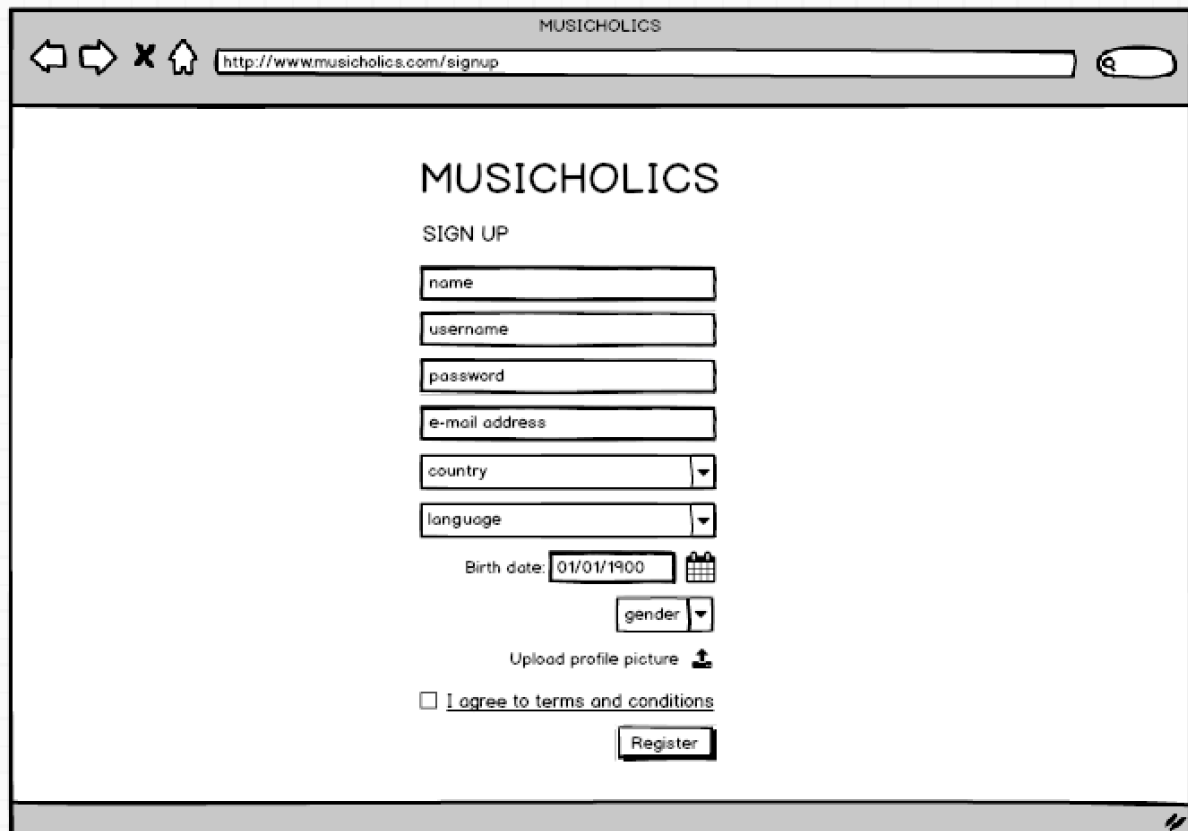
SQL Statements

```
SELECT username, password
```

```
FROM Person
```

```
WHERE username = @username AND password = @password
```

5.3. Register



The screenshot shows a web browser window with the title "MUSICHOLICS". The address bar displays "http://www.musicholics.com/signup". The page content features the "MUSICHOLICS" logo at the top, followed by the heading "SIGN UP". Below this heading is a registration form with the following fields: "name", "username", "password", "e-mail address", "country" (a dropdown menu), "language" (a dropdown menu), "Birth date:" (a text box containing "01/01/1900" and a calendar icon), "gender" (a dropdown menu), and "Upload profile picture" (a text label with an upload icon). At the bottom of the form is a checkbox labeled "I agree to terms and conditions" and a "Register" button.

Inputs: @fullname, @username, @password, @email, @country, @language, @birthday, @gender, @picture

Process: The user signs up to use Musicholics. She enters her name, username, password, e-mail address, country, language, birth date, and gender to be able to use the system. Also, she upload her picture.

SQL Statements

INSERT INTO User

VALUES(@username, @fullname, @password, @email, @country, @language, @picture, GETDATE(), @birthday, @gender, 0)

5.4. Search Result Screen

5.4.1. Search Result Screen for Track



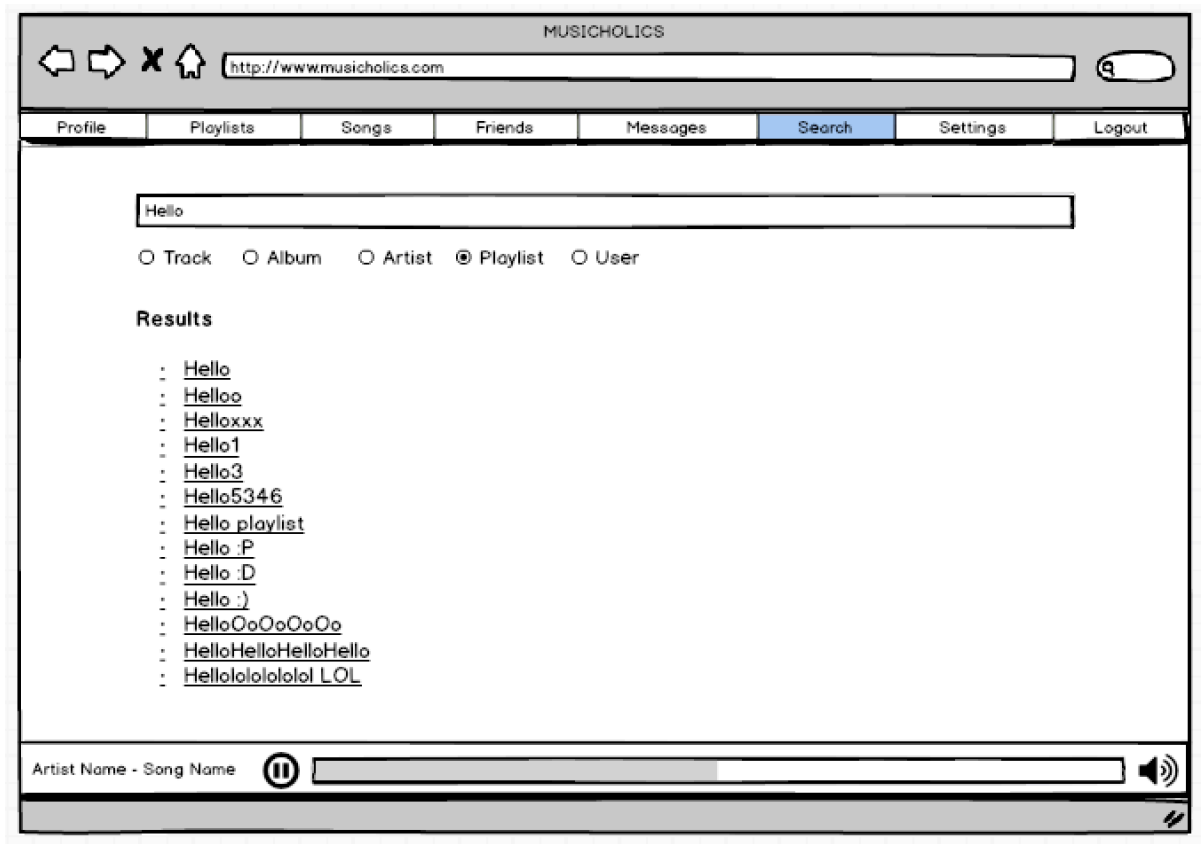
Inputs: @searchquery

Process: The user can search for a track and the system displays the results of the search query.

SQL Statements

No SQL Statement is needed because KMP algorithm will be used to search keyword in the data.

5.4.2. Search Result Screen for Playlist



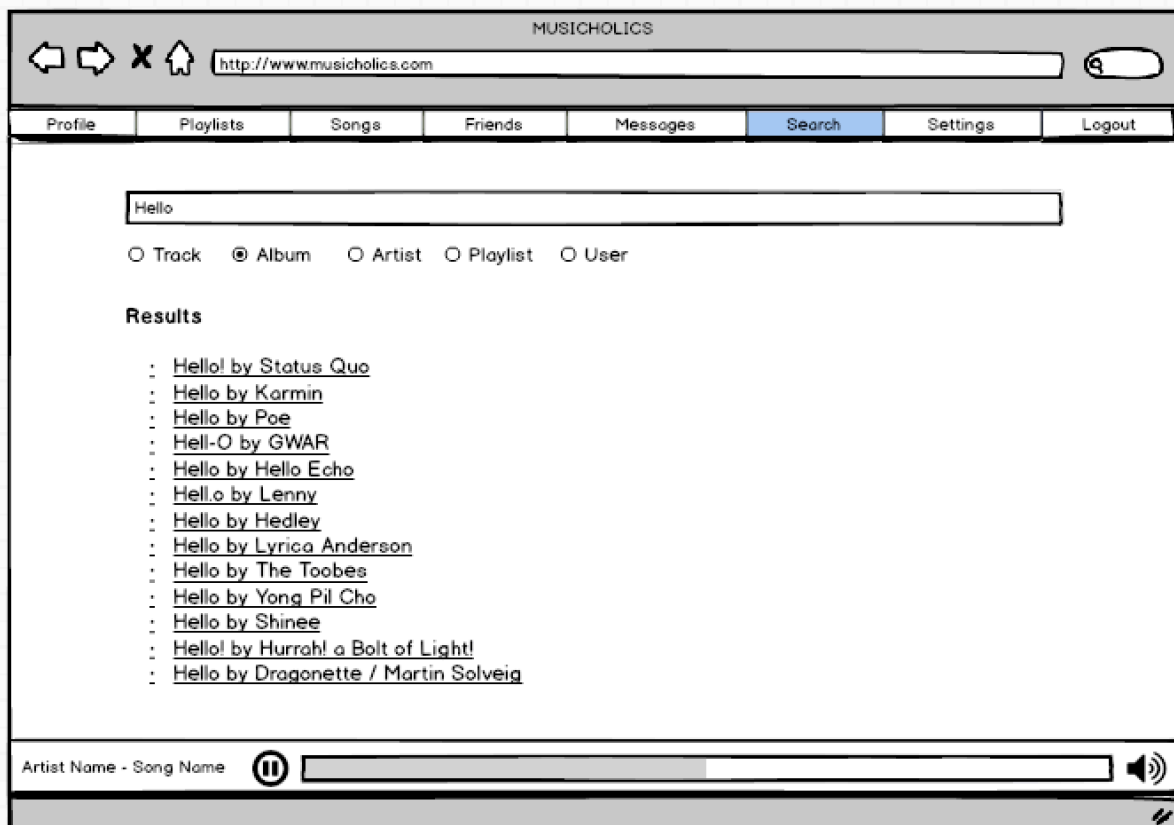
Inputs: @searchquery

Process: The user can search for a playlist and the system displays the results of the search query.

SQL Statements

No SQL Statement is needed because KMP algorithm will be used to search keyword in the data.

5.4.3. Search Result Screen for Album



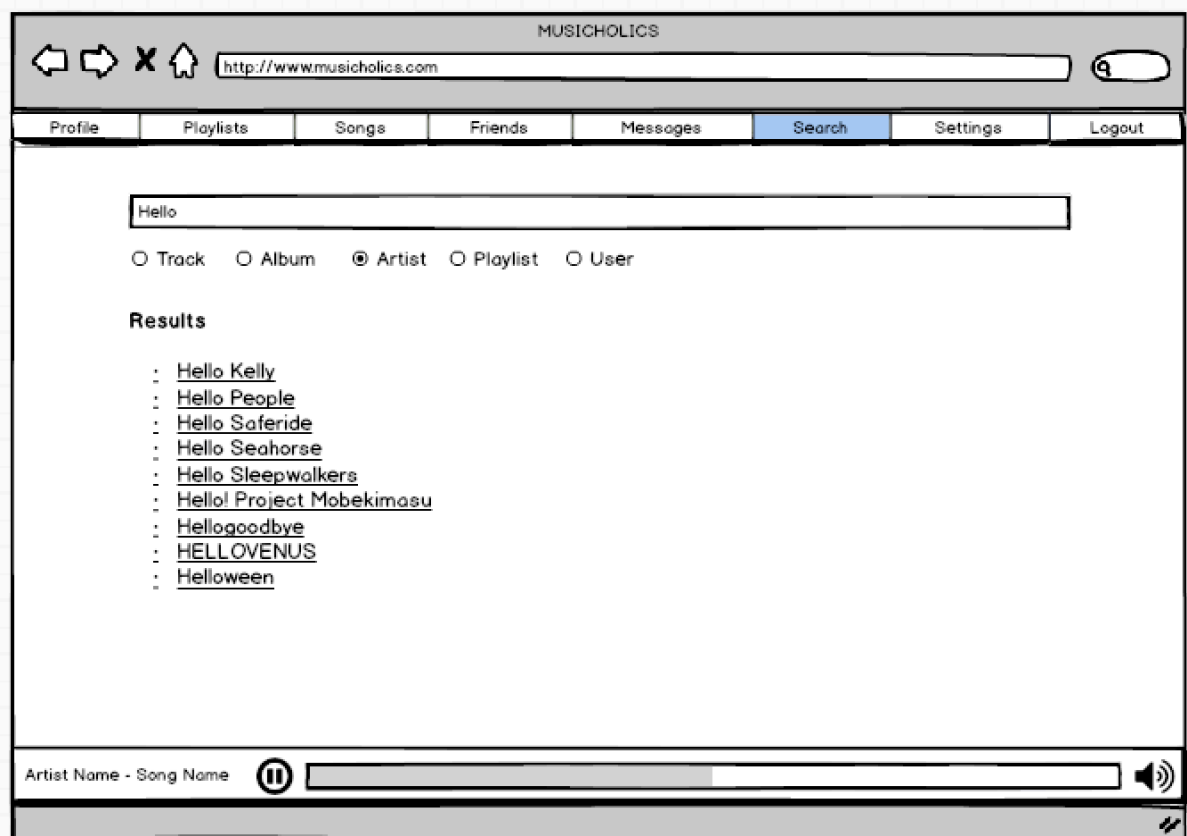
Inputs: @searchquery

Process: The user can search for a album and the system displays the results of the search query.

SQL Statements

No SQL Statement is needed because KMP algorithm will be used to search keyword in the data.

5.4.4. Search Result Screen for Artist



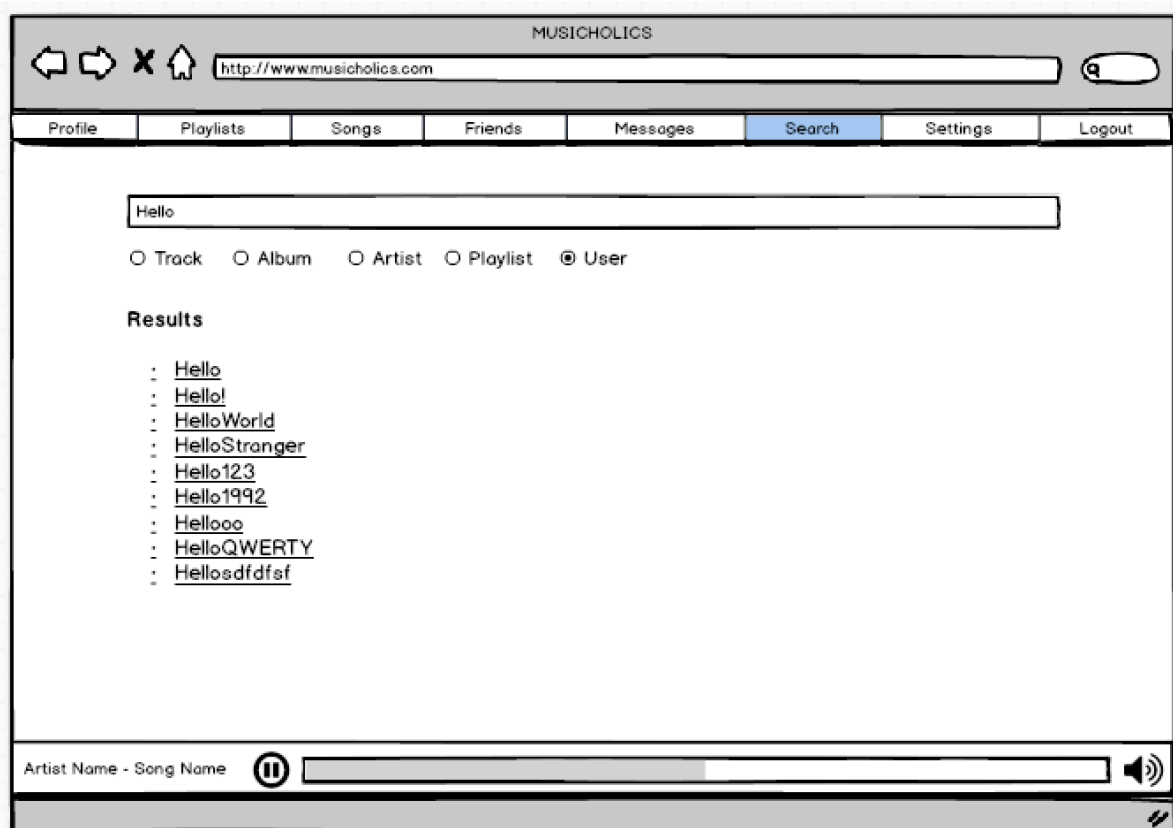
Inputs: @searchquery

Process: The user can search for a artist and the system displays the results of the search query.

SQL Statements

No SQL Statement is needed because KMP algorithm will be used to search keyword in the data.

5.4.5. Search Result Screen for User



Inputs: @searchquery

Process: The user can search for a person and the system displays the results of the search query.

SQL Statements

No SQL Statement is needed because KMP algorithm will be used to search keyword in the data.

5.5. Change Personal Information

5.5.1. Change General Information

The screenshot shows a web browser window titled 'MUSICHOLOGICS' with the address bar displaying 'http://www.musicholics.com'. The navigation bar includes links for Profile, Playlists, Songs, Friends, Messages, Search, Settings (highlighted), and Logout. The main content area is titled 'Change personal information' and features a user profile icon with an 'Update' button. The form contains the following fields: Name (Jane Doe), E-mail address (janedoe@ug.bilkent.edu.tr), Country (Turkey), Language (English), Birth date (01/01/1900), Gender (Female), Membership Type (Free), and an option to 'UPGRADE with budget or credit card'. A 'Password: Change' link is also present. A checkbox for 'I am aware that changes might not be recovered.' is located below the password field, followed by an 'Apply' button. The bottom of the page features a music player interface with 'Artist Name - Song Name' and a progress bar.

Inputs: @person_id, @new_fullname, @new_email, @new_country, @new_language, @new_birthday, @new_gender

@person_id is id of the user logged in.

Process: Logged in users can change their personal information from “Settings” menu. They are able to change their name, e-mail address, country, language, birth date, and gender. Also, password is changeable too.

SQL Statements

View Information:

```
SELECT fullname, email, country, language, birthday, gender, membership_type
FROM Person
WHERE person_id = @person_id
```

Change Fullname:

```
UPDATE Person
SET fullname = @new_fullname
WHERE person_id = @person_id
```

Change Email:

```
UPDATE Person
SET email = @new_email
WHERE person_id = @person_id
```

Change Country:

```
UPDATE Person
SET country = @new_country
WHERE person_id = @person_id
```

Change Language:

```
UPDATE Person
SET language = @new_language
WHERE person_id = @person_id
```

Change Birthdate:

```
UPDATE Person
SET birthday = @new_birthday
WHERE person_id = @person_id
```

Change Gender:

```
UPDATE Person
SET gender = @new_gender
WHERE person_id = @person_id
```

5.5.2. Change Password

The screenshot shows a web browser window with the URL <http://www.musicholics.com>. The browser's address bar and the website's navigation menu are visible. The navigation menu includes links for Profile, Playlists, Songs, Friends, Messages, Search, Settings (which is highlighted), and Logout. The main content area is titled "Change password" and contains three input fields labeled "old password", "new password", and "re-enter new password". Below these fields is a checkbox with the text "I understand this cannot be recovered." and an "Apply" button. At the bottom of the page, there is a music player interface showing "Artist Name - Song Name" and a progress bar.

Inputs: @person_id, @oldPassword, @newPassword, @newPasswordAgain

@person_id is id of the person (admin or user) signed in.

Process: The person (admin or user) can change her password. This menu can be reached from another menu called "Change Personal Information".

SQL Statements

```
SELECT password
```

```
FROM Person
```

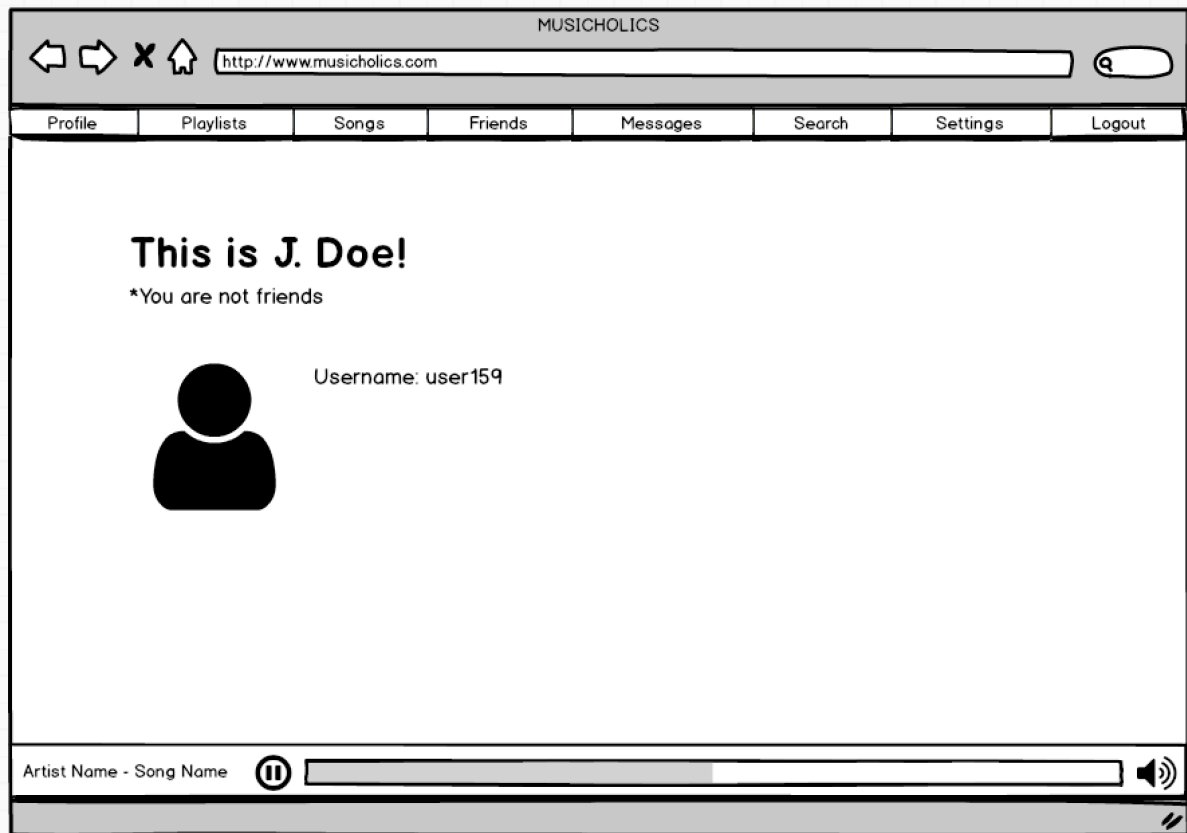
```
WHERE person_id = @person_id
```

If @newPassword matches with @newPassword2 AND @oldPassword matches in PHP
update password:

```
UPDATE Person SET password = @newPassword WHERE person_id = @person_id
```

5.6. Profile

5.6.1. Blocked Profile



Inputs: @user_id

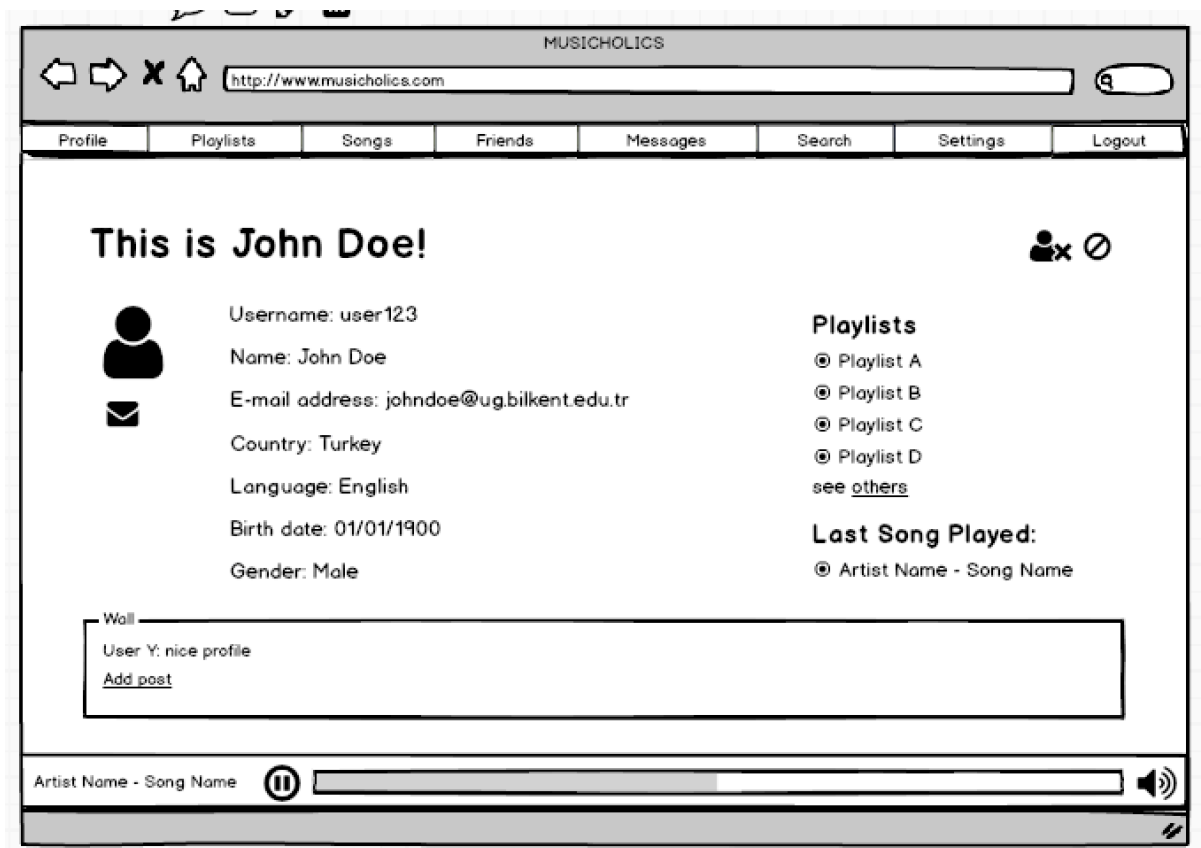
@user_id is id of displayed blocked user.

Process: The user can view a user who is blocked by her.

SQL Statements

```
SELECT username, fullname
FROM User
WHERE person_id = @user_id
```

5.6.2. Friend Profile



Inputs: @person_id, @user_id, @post

@person_id is id of the user logged in. @user_id is id of displayed user. @post is post added by the user logged in.

Process: The user can view her friend's profile.

SQL Statements

Show Information:

```
SELECT username, fullname, email, country, language, birthday, gender
FROM User
WHERE person_id = @user_id
```

Remove from Friendship:

```
DELETE FROM Friendship
```

```
WHERE user1_id = @person_id AND user2_id = @user_id
```

```
DELETE FROM Friendship
```

```
WHERE user2_id = @person_id AND user1_id = @user_id
```

Block User:

```
INSERT INTO Blocks
```

```
VALUES( @person_id , @user_id )
```

View Playlists:

```
SELECT playlist_name
```

```
FROM Playlist
```

```
WHERE playlist_id = @user_id
```

View Track:

```
SELECT T.track_name
```

```
FROM Track T, Listens L
```

```
WHERE L.user_id = @user_id AND L.track_id = T.track_id AND
```

```
      L.date = (      SELECT MIN( L2.date )
```

```
                    FROM Track T2, Listens L2
```

```
                    WHERE L2.user_id = @user_id AND L2.track_id = T2.track_id )
```

View Wall:

```
SELECT writer_id, post
```

```
FROM Posts
```

```
WHERE receiver_id = @user_id
```

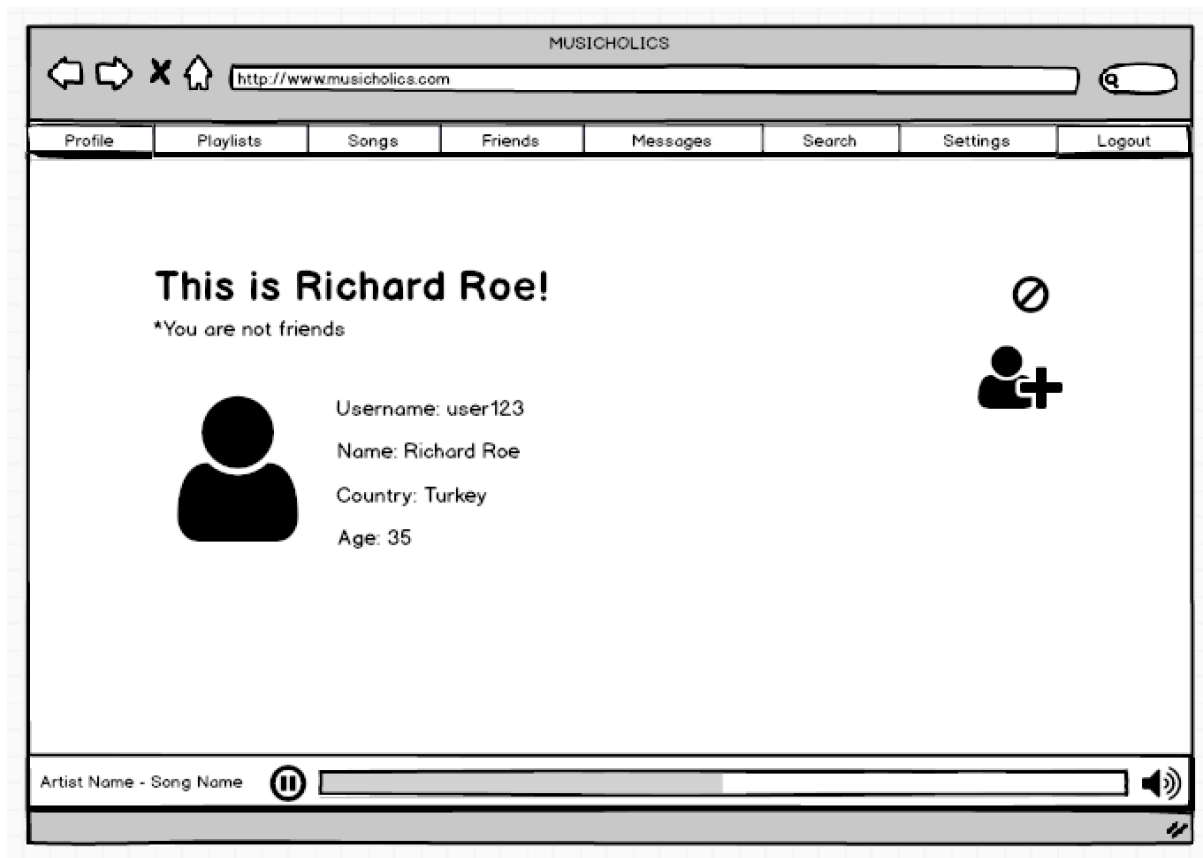
```
ORDER BY date
```

Post to Wall:

INSERT INTO Posts

VALUES(@person_id, @user_id, GETDATE(), @post)

5.6.3. Nonfriend Profile



Inputs: @user_id

@user_id is person_id of displayed user.

Process: The user can view her non-friend user profile.

SQL Statements

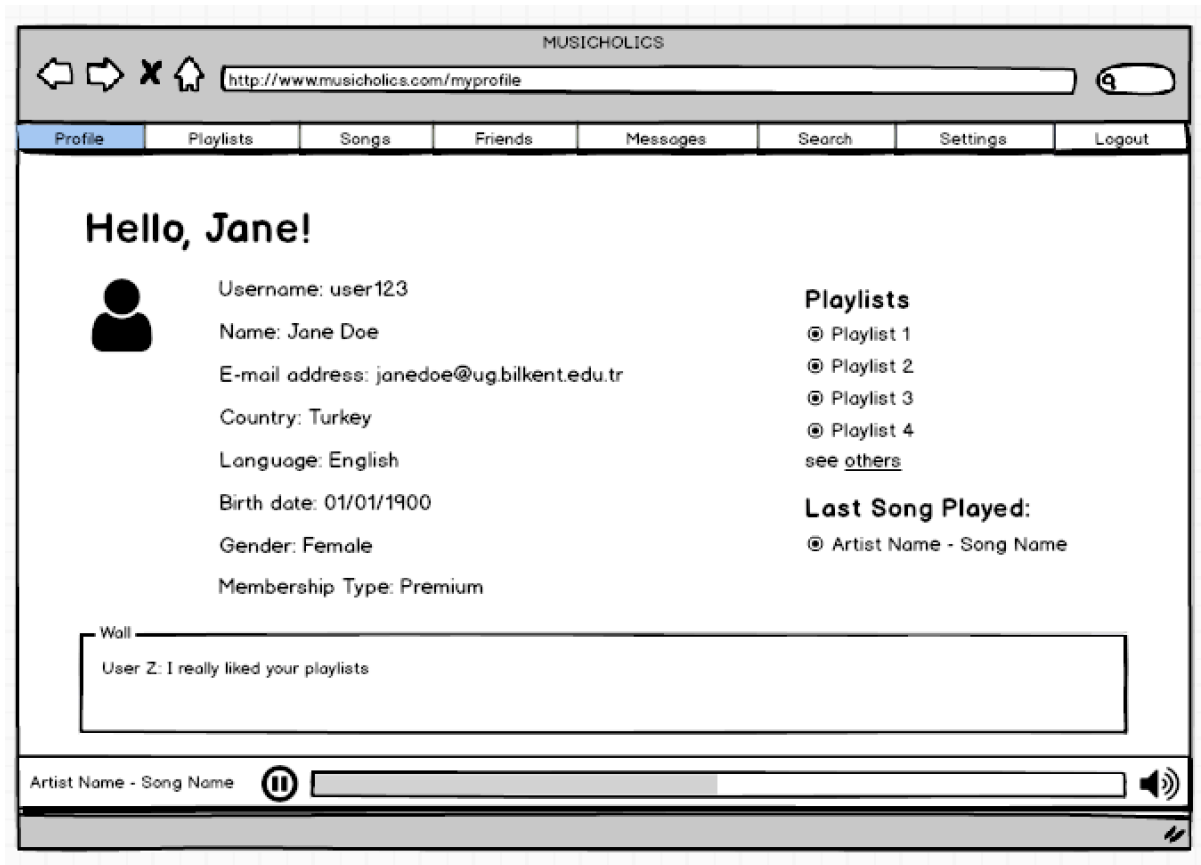
View Information:

```
SELECT username, fullname, country  
FROM User  
WHERE person_id = @user_id
```

Add Friend:

```
INSERT INTO Friendship  
VALUES( @person_id , @user_id )
```


5.6.4. Own Profile



Inputs: @person_id

@person_id is id of the user logged in.

Process: The user can view her own profile from "Profile" section.

SQL Statements

Almost all of SQL is similar to 5.6.2. where @person_id should be used instead of @user_id.

Show Information:

```
SELECT username, fullname, email, country, language, birthday, gender, membership_type
FROM User
WHERE person_id = @person_id
```

5.6.5. Complete Profile



Inputs: @person_id, @user_id

@person_id is id of the admin logged in. @user_id is id of displayed user.

Process: The admin can view any user profile without any restriction.

SQL Statements

View Information:

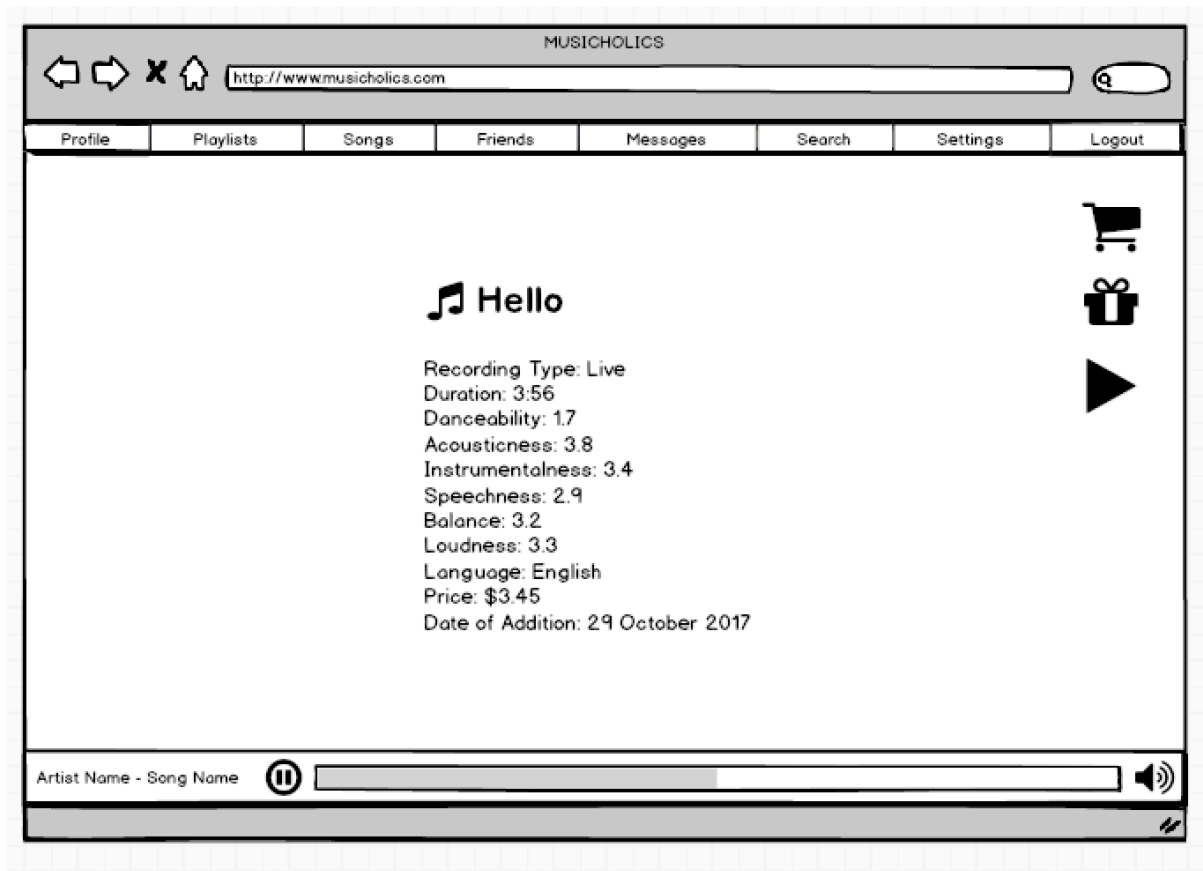
```
SELECT username, fullname, email, country, language, birthday, gender, membership_type
FROM User
WHERE person_id = @user_id
```

Ban User:

```
INSERT INTO Bans
VALUES( @user_id , @person_id )
```

5.7. Track

5.7.1. View Track



Inputs: @track_id

@person_id is id of the user logged in. @track_id is id of the track that is viewed by the user.
@receiver_id is id of the receiver of the gift.

Process: The user can view any track. She can reach its information including its name, recording type, duration, danceability, acousticness, instrumentalness, speechness, balance, loudness, language, price, and date of addition to the system. The user is able to play this track, buy it, and gift it to another user.

SQL Statements

View Information:

```
SELECT track_name, recording_type, duration, danceability, acousticness, instrumentalness,  
speechness, balance, loudness, language, price, date_of_addition
```

```
FROM Track
```

```
WHERE track_id = @track_id
```

Add to Card:

```
INSERT INTO Buys
```

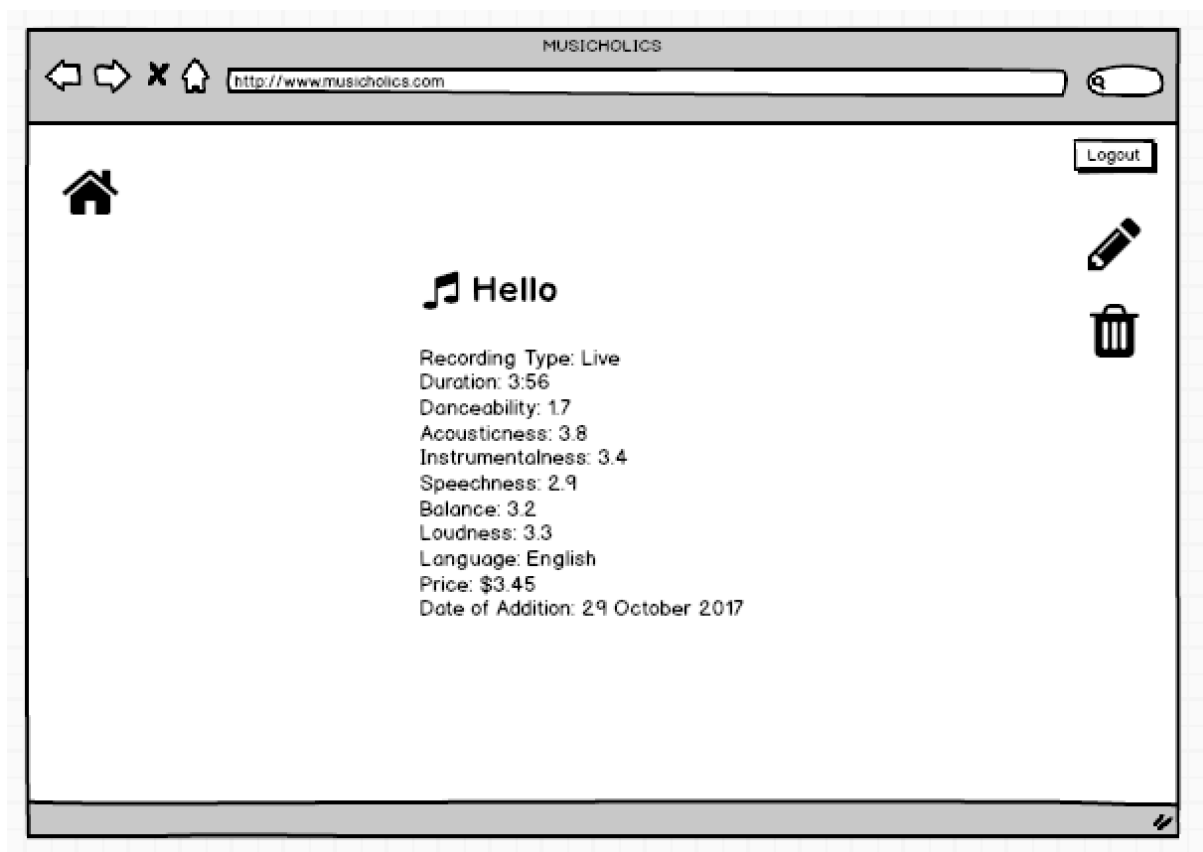
```
VALUES( @person_id, @track_id )
```

Gift the Track:

```
INSERT INTO Gift
```

```
VALUES( @person_id, @receiver_id, @track_id )
```

5.7.2. Access Track



Inputs: @track_id

Process: The Admin can access any track. She can reach its information including its name, recording type, duration, danceability, acousticness, instrumentalness, speechness, balance, loudness, language, price, and date of addition to the system. The admin can modify and delete this track.

SQL Statements

View Information:

```
SELECT track_name, recording_type, duration, danceability, acousticness, instrumentalness,
speechness, balance, loudness, language, price, date_of_addition
```

```
FROM Track
```

```
WHERE track_id = @track_id
```

Remove Track:

DELETE FROM Track

WHERE track_id = @track_id

DELETE FROM Track_Belongs_To_Artist

WHERE track_id = @track_id

DELETE FROM Added

WHERE track_id = @track_id

DELETE FROM Track

WHERE track_id = @track_id

DELETE FROM Buys

WHERE track_id = @track_id

DELETE FROM Gift

WHERE track_id = @track_id

DELETE FROM Listens

WHERE track_id = @track_id

5.7.3. Modify Track

The screenshot shows a web browser window titled 'MUSICHOOLICS' with the URL 'http://www.musicchoolics.com'. The page has a home icon and a 'Logout' button in the top right. The main heading is 'Change track information'. The form contains the following fields:

- Recording Type:
- Duration:
- Danceability:
- Acousticness:
- Instrumentalness:
- Speechness:
- Balance:
- Loudness:
- Language:
- Price: \$
- Date of Addition:

Inputs: @track_id, @new_recording_type, @new_duration, @new_danceability, @new_acousticness, @new_instrumentalness, @new_speechness, @new_balance, @new_loudness, @new_language, @new_price, @new_date_of_addition

@track_id is the id of track that the admin is looking for.

Process: The admin can change information of any track.

SQL Statements

View Information:

```
SELECT recording_type, duration, danceability, acousticness, instrumentalness, speechness,
balance, loudness, language, price, date_of_addition
FROM Track
WHERE track_id = @track_id
```

Change Recording Type:

```
UPDATE Track
SET recording_type = @new_recording_type
WHERE track_id = @track_id
```

Change Duration:

```
UPDATE Track
SET duration = @new_duration
WHERE track_id = @track_id
```

Change Danceability:

```
UPDATE Track
SET danceability = @new_danceability
WHERE track_id = @track_id
```

Change Acousticness:

```
UPDATE Track
SET acousticness = @new_acousticness
WHERE track_id = @track_id
```

Change Instrumentalness:

```
UPDATE Track
SET instrumentalness = @new_instrumentalness
WHERE track_id = @track_id
```

Change Speechness:

```
UPDATE Track
SET speechness = @new_speechness
WHERE track_id = @track_id
```


Change Balance:

```
UPDATE Track  
SET balance = @new_balance  
WHERE track_id = @track_id
```

Change Loudness:

```
UPDATE Track  
SET loudness = @new_loudness  
WHERE track_id = @track_id
```

Change Language:

```
UPDATE Track  
SET language = @new_language  
WHERE track_id = @track_id
```

Change Price:

```
UPDATE Track  
SET price = @new_price  
WHERE track_id = @track_id
```

Change Date of Addition:

```
UPDATE Track  
SET date_of_addition = @new_date_of_addition  
WHERE track_id = @track_id
```

5.8. Publisher

The screenshot shows a web browser window titled "MUSICHOLICS" with the URL "http://www.musicholics.com". The page features a home icon in the top left and a "Logout" button in the top right. The main content area displays the title "Publisher XXX". Below the title are two dropdown menus: "Country:" with "Turkey" selected and "City:" with "Ankara" selected. A checkbox labeled "I am aware that changes might not be recovered." is present, followed by an "Apply" button. A small double-slash icon is visible in the bottom right corner of the page frame.

Inputs: @publisher_id, @new_country, @new_city

@publisher_id is the id of publisher that the admin is looking for.

Process: The admin can access any publisher, and change the information related to that.

SQL Statements

View Information:

```
SELECT publisher_name, country, city
FROM Publisher
WHERE publisher_id = @publisher_id
```

Change Country:

UPDATE Publisher

SET country = @new_country

WHERE publisher_id = @publisher_id

Change City:

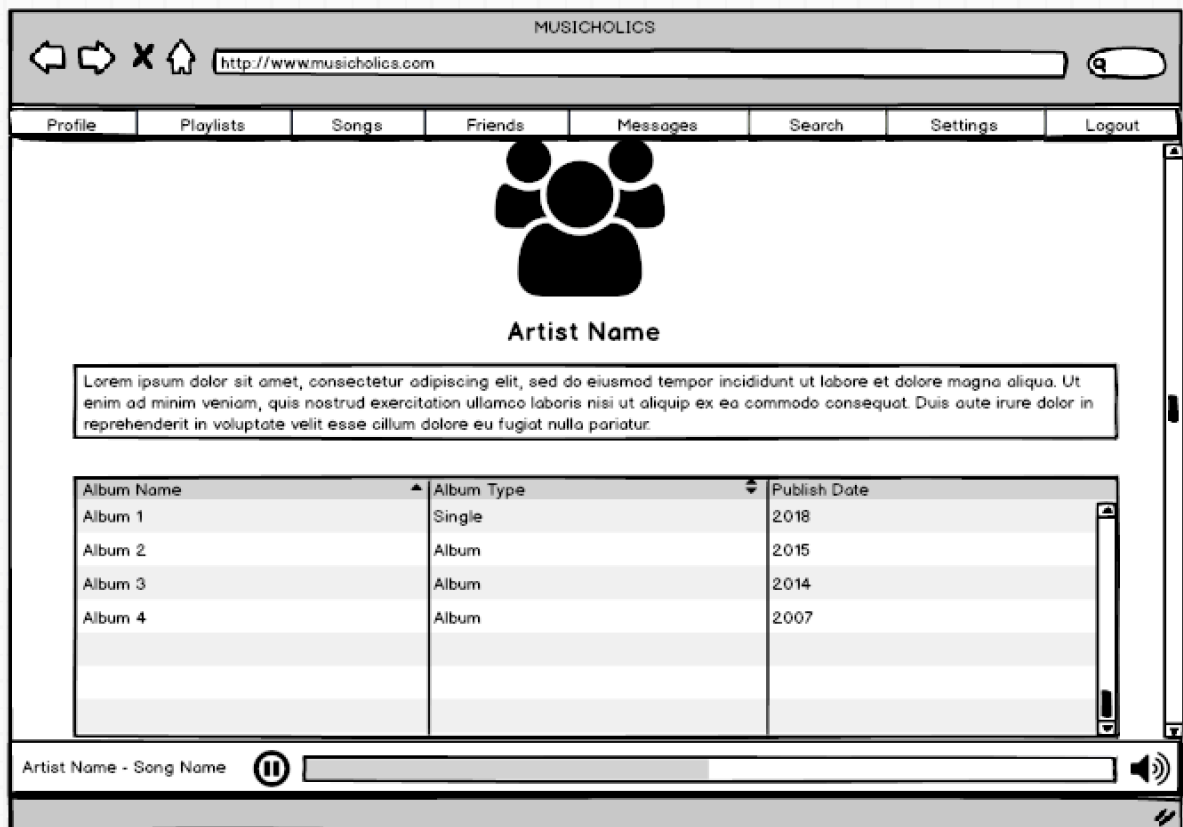
UPDATE Publisher

SET city = @new_city

WHERE publisher_id = @publisher_id

5.9. Artist

5.9.1. View Artist



Inputs: @artist_id

@artist_id is id of the artist who is viewed by the user.

Process: The user can view any artist and access her albums. For each album, type and publish date of the album is specified.

SQL Statements

View Artist Info:

```
SELECT artist_name, picture, description
FROM Artist
WHERE artist_id = @artist_id
```

View Albums:

```
SELECT ALB.album_name, ALB.album_type, ALB.published_date
FROM Album_Belongs_To_Artist ABTA, Album ALB
WHERE ABTA.artist_id = @artist_id AND
      ABTA.album_id = ALB.album_id
```

5.9.2. Modify Artist

Album Name	Album Type	Publish Date	Select
Album 1	Single	2018	<input type="checkbox"/>
Album 2	Album	2015	<input type="checkbox"/>
Album 3	Album	2014	<input type="checkbox"/>

Add Album:

Inputs: @artist_id, @album_id

@artist_id is id of the artist who is viewed by the admin.

@album_id is id of the album which is entered by the admin.

Process: The admin can view any artist and access her albums. For each album, type and publish date of the album is specified. Also, she is able to add existing albums.

SQL Statements

Similar SQL Statements to the user

Add album:

```
INSERT INTO Album_Belongs_To_Artist
VALUES(@artist_id, @album_id)
```

Track Info:

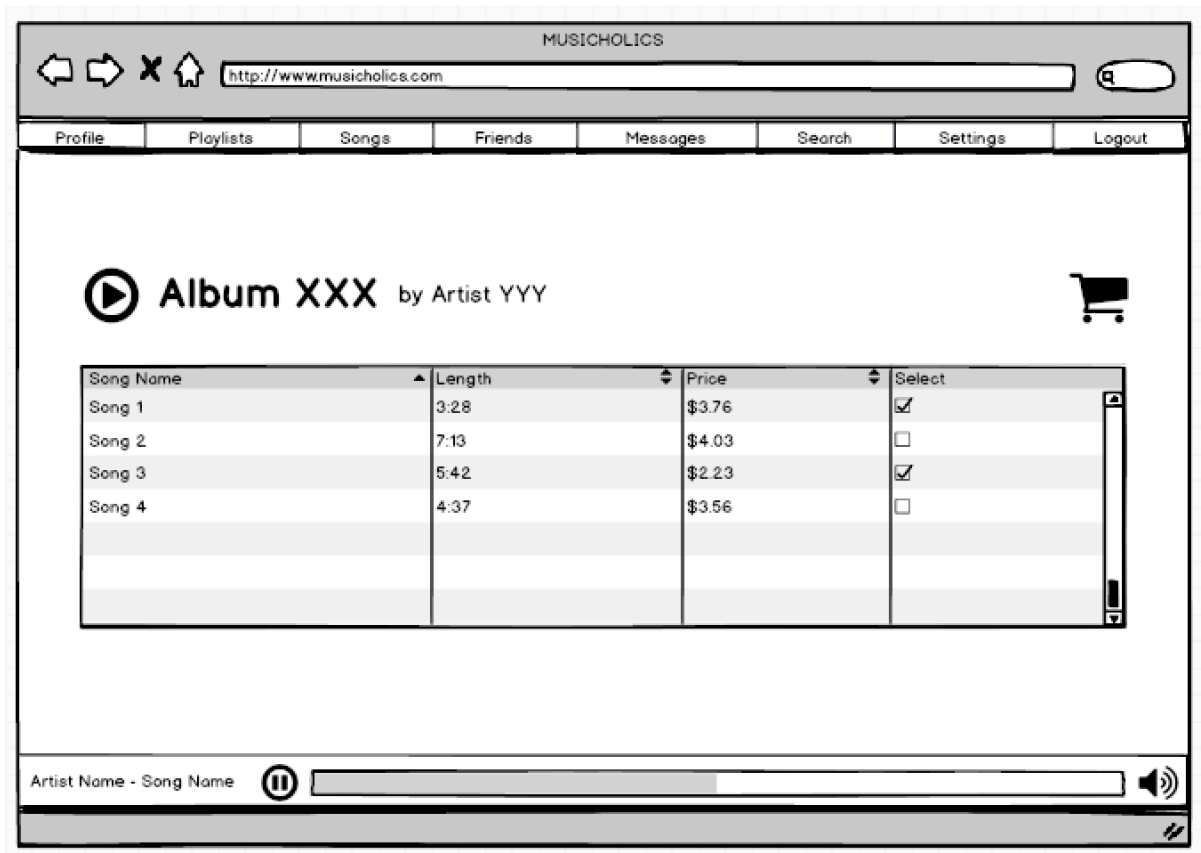
```
SELECT album_name, album_type, published_date  
FROM Album  
WHERE album_id = @album_id
```

Remove Artist:

```
DELETE FROM Album  
WHERE artist_id = @artist_id  
DELETE FROM Album_Belongs_To_Artist  
WHERE artist_id = @artist_id  
DELETE FROM Track_Belongs_To_Artist  
WHERE artist_id = @artist_id
```

5.10.Album

5.10.1. View Album



Inputs: @album_id

@album_id is the id of the album which is viewed by the user.

Process: The user can view an album. She is also able to view artists, and tracks related to the album. Moreover, she can play the album.

SQL Statements

View Album Info:

```
SELECT album_name
FROM Album
WHERE album_id = @album_id
```


View Artists:

```
SELECT AR.artist_name  
FROM Album_Belongs_To_Artist ABTA, Artist AR  
WHERE ABTA.album_id = @album_id AND ABTA.artist_id = AR.artist_id
```

View Tracks:

```
SELECT T.track_name, T.duration, T.price  
FROM Track T  
WHERE T.album_id = @album_id
```

5.10.2. Modify Album

Song Name	Length	Price
Song 1	3:28	\$3.76
Song 2	7:13	\$4.03
Song 3	5:42	\$2.23
Song 4	4:37	\$3.56

Add Song:

Inputs: @album_id, @track_id

@album_id is the id of the album which is viewed by the admin.

@track_id is the id of the track which is entered by the admin.

Process: The admin can add tracks to the album. When the admin wants to enter an existing track, she enters the id of it. Then, some information of it appears.

SQL Statements

Similar SQL Statements to the user

Add Track:

UPDATE TRACK

SET album_id = @album_id

WHERE track_id = @track_id

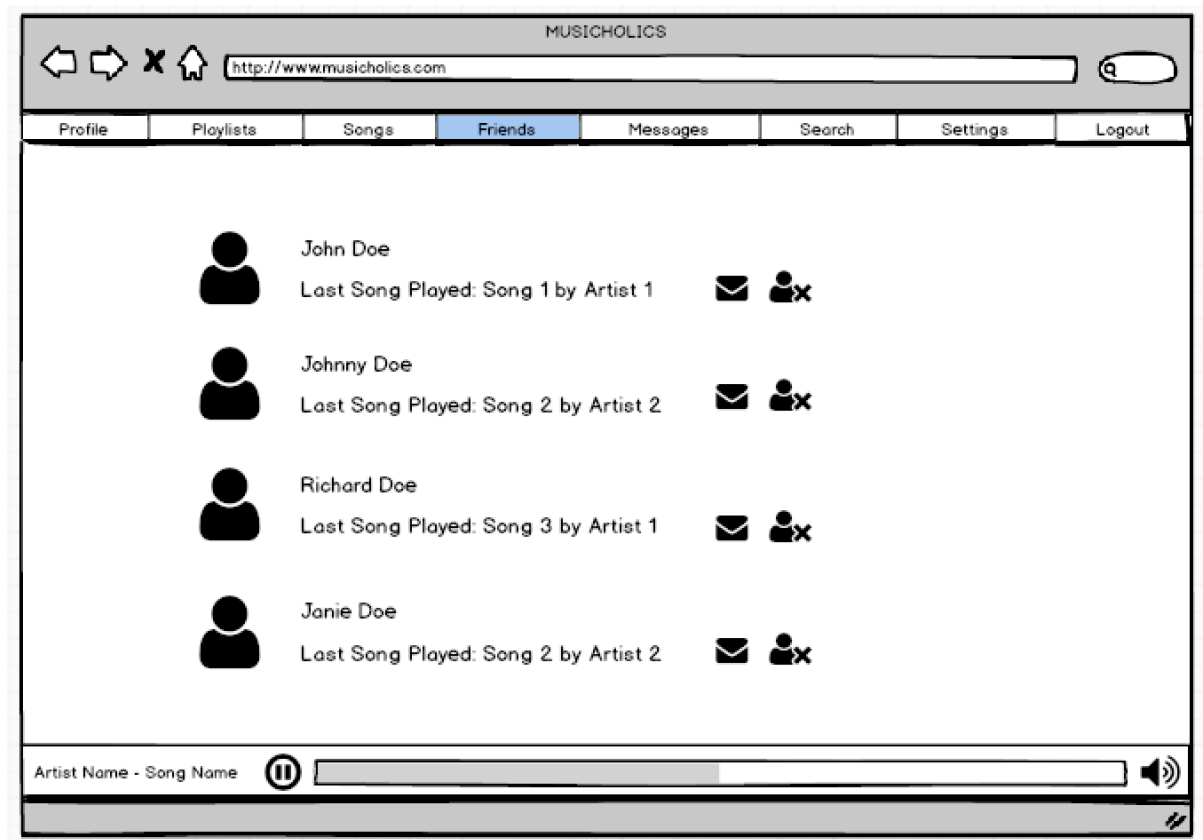
Track Info:

```
SELECT track_name, duration, price  
FROM Track  
WHERE track_id = @track_id
```

Remove Album:

```
DELETE FROM Album  
WHERE album_id = @album_id  
DELETE FROM Track  
WHERE album_id = @album_id  
DELETE FROM Album_Belongs_To_Artist  
WHERE album_id = @album_id
```

5.11.Friends List



Inputs: @person_id, @friend_id

@person_id is id of the user logged in. @friend_id is id of the friend to be removed.

Process: The user can view her friends. Also, she is able to send a message or remove from the friend list.

SQL Statements

View Friends:

```
SELECT U.fullname, U.picture
FROM User U, Friendship F
WHERE U.person_id = F.user1_id AND @person_id = F.user2_id
UNION
SELECT U.fullname, U.picture
FROM User U, Friendship F
WHERE U.person_id = F.user1_id AND @person_id = F.user2_id
```

Remove Friend:

```
DELETE FROM Friendship
```

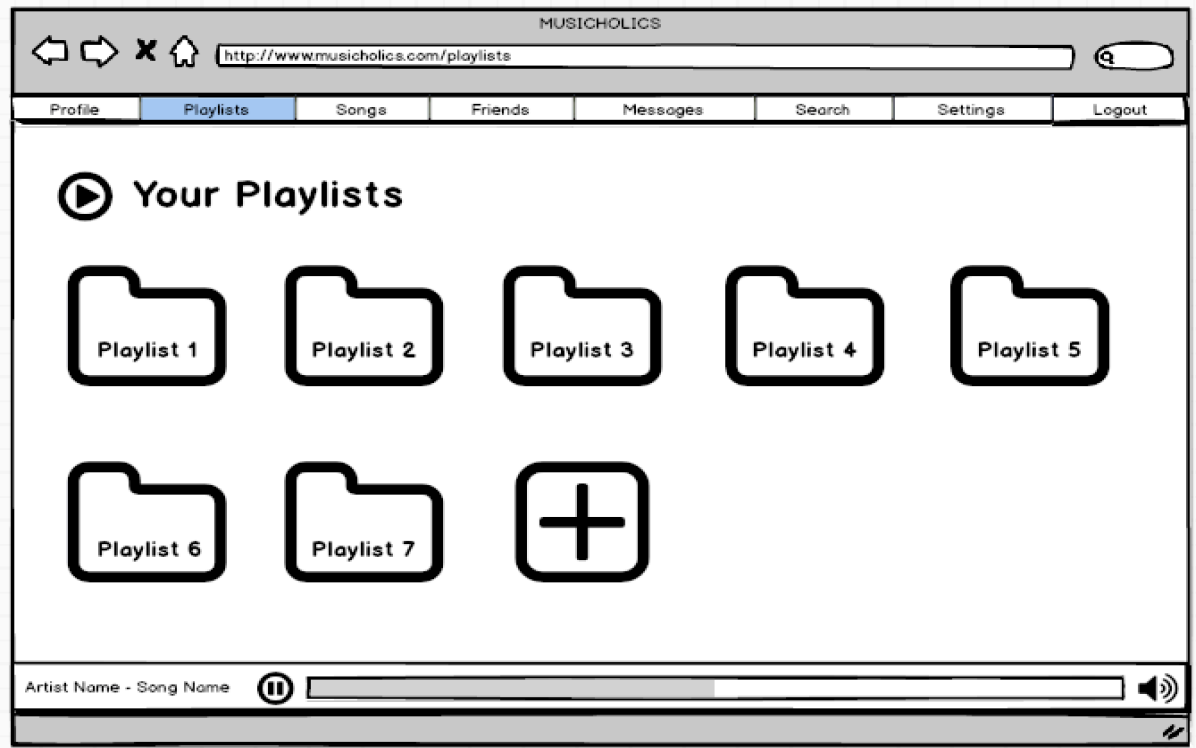
```
WHERE user1_id = @person_id AND user2_id = @friend_id
```

```
DELETE FROM Friendship
```

```
WHERE user2_id = @person_id AND user1_id = @friend_id
```

5.12.Playlist

5.12.1. View Playlists



Inputs: @person_id, @playlist_name, @description, @picture

@person_id is id of the user logged in.

Process: The user can view her own playlists. Also, she can create a new playlist.

SQL Statements:

View Playlists:

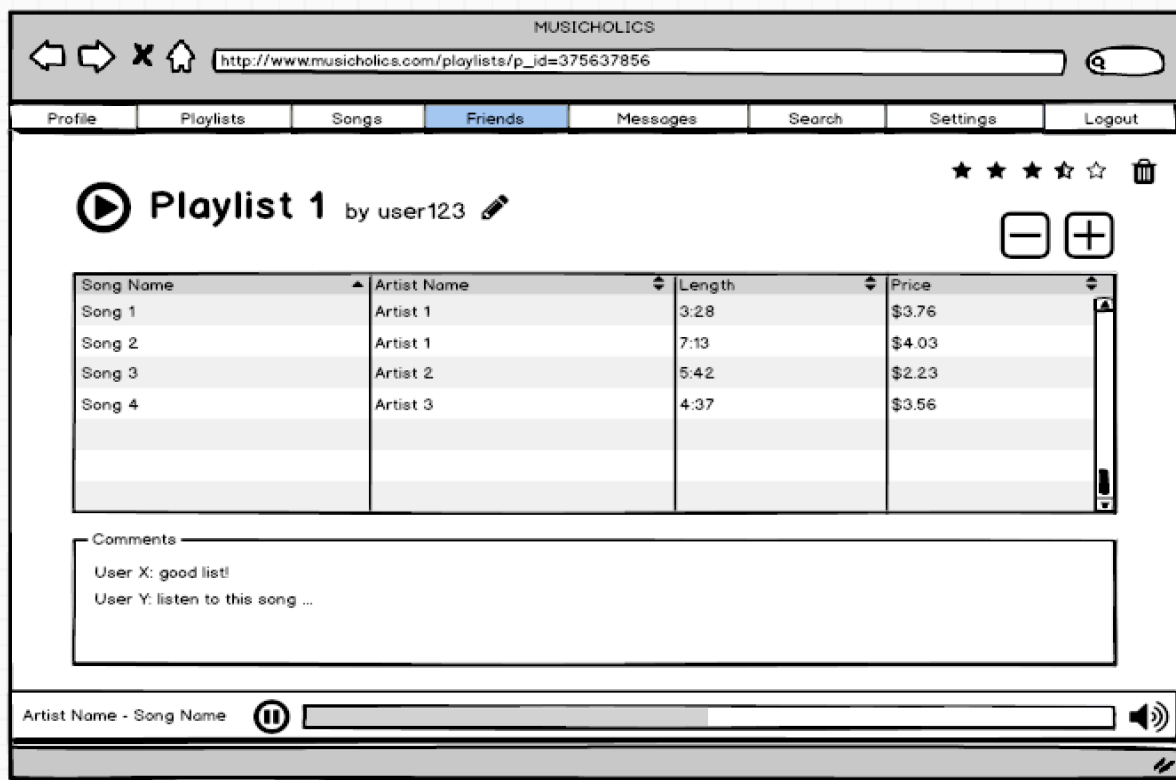
```
SELECT P.playlist_name
FROM Playlist P, Follows F
WHERE F.user_id = @person_id AND F.playlist_id = P.playlist_id
UNION
SELECT playlist_name
FROM Playlist
WHERE creator_id = @person_id
```

Add Playlist:

INSERT INTO Playlist

VALUES(@playlist_name , @description , @picture , @person_id)

5.12.2. View Own Playlist



Inputs: @playlist_id, @track_id

@playlist_id is id of the playlist that the user is looking. @track_id is id of the track to be removed or added.

Process: The user can view her one of the playlist.

SQL Statements:

View Playlist Info:

```
SELECT playlist_name
FROM Playlist
WHERE playlist_id = @playlist_id
```

View Tracks:

```
SELECT T.track_name, T.duration, T.price
FROM Added A, Track T
WHERE A.playlist_id = @playlist_id AND T.track_id = A.track_id
```


Remove Playlist:

```
DELETE FROM Playlist
WHERE playlist_id = @playlist_id
DELETE FROM Added
WHERE playlist_id = @playlist_id
DELETE FROM Follows
WHERE playlist_id = @playlist_id
DELETE FROM Rates
WHERE playlist_id = @playlist_id
DELETE FROM Comments
WHERE playlist_id = @playlist_id
DELETE FROM Collaborates
WHERE playlist_id = @playlist_id
```

Remove Track:

```
DELETE FROM Added
WHERE playlist_id = @playlist_id AND track_id = @track_id
```

Add Track:

```
INSERT INTO Added
VALUES( @playlist_id , @track_id , GETDATE() )
```

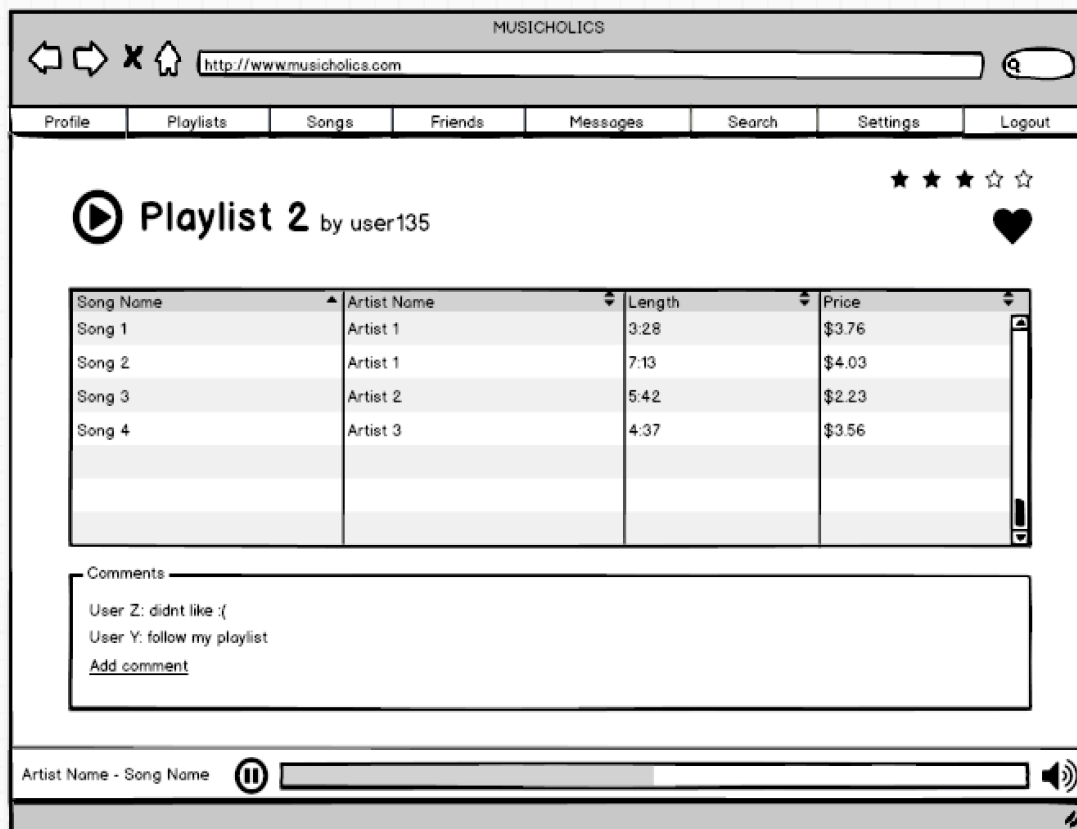
Get Rate:

```
SELECT AVG( rate )
FROM Rates
WHERE @playlist_id = playlist_id
```

View Comments:

```
SELECT user_id, comment  
FROM Comments  
WHERE @playlist_id = playlist_id  
ORDER BY date
```

5.12.3.View Another User's Playlist



Inputs: @person_id , @playlist_id, @rate, @comment

@person_id is id of the user logged in. @playlist_id is id of the playlist that the user is looking. @rate is given by the user logged in. @comment is made by the user logged in.

Process: The user can view a playlist of another user.

SQL Statements:

An important part of the SQL code is similar to 5.12.2.

View Playlist Info:

```
SELECT playlist_name
FROM Playlist
WHERE playlist_id = @playlist_id
```

View Tracks:

```
SELECT T.track_name, T.duration, T.price  
FROM Added A, Track T  
WHERE A.playlist_id = @playlist_id AND T.track_id = A.track_id
```

Follow Playlist:

```
INSERT INTO Follows  
VALUES( @person_id , @playlist_id )
```

Collaborate Playlist:

```
INSERT INTO Collaborates  
VALUES( @person_id , @playlist_id )
```

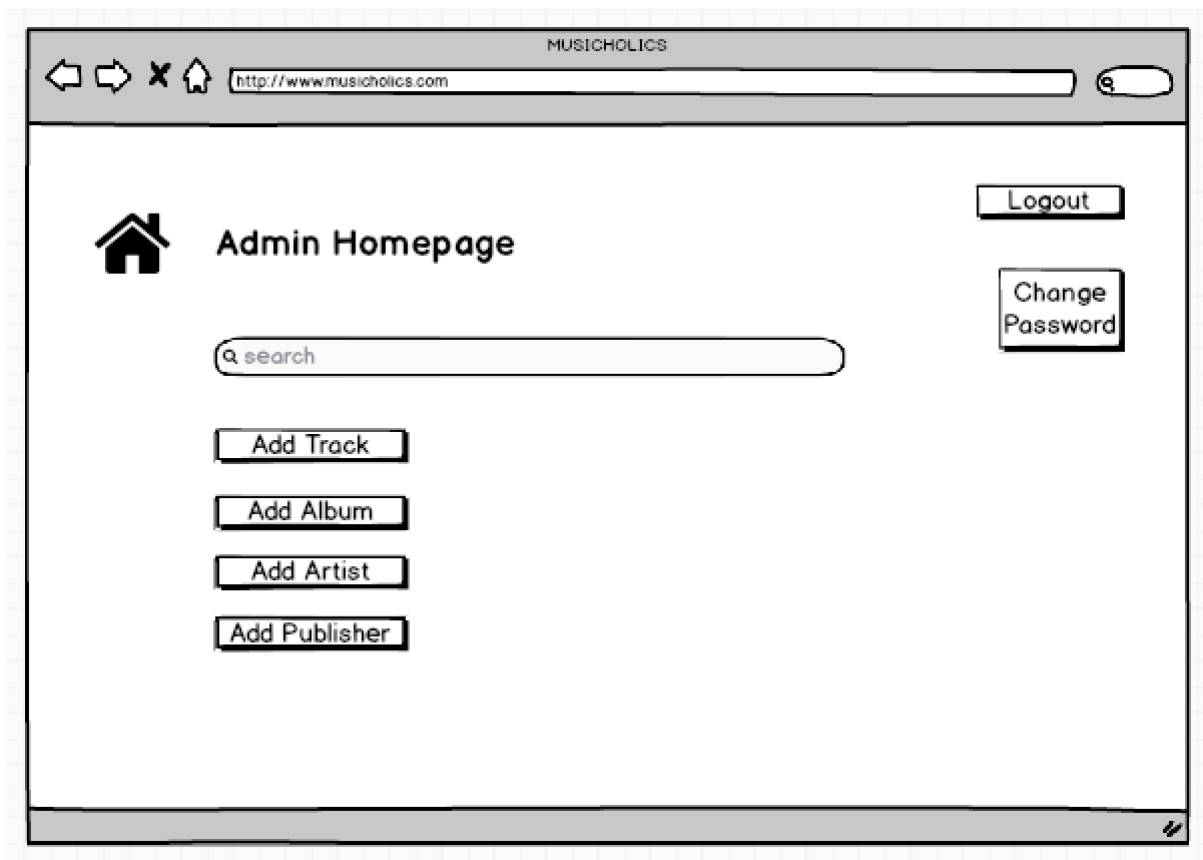
Rate Playlist:

```
INSERT INTO Rates  
VALUES( @person_id , @playlist_id , @rate )
```

Comment Playlist:

```
INSERT INTO Comments  
VALUES( @person_id , @playlist_id , GETDATE() , @comment )
```

5.13.Admin Panel



Inputs: @searchquery

@searchquery is the query requested by admin.

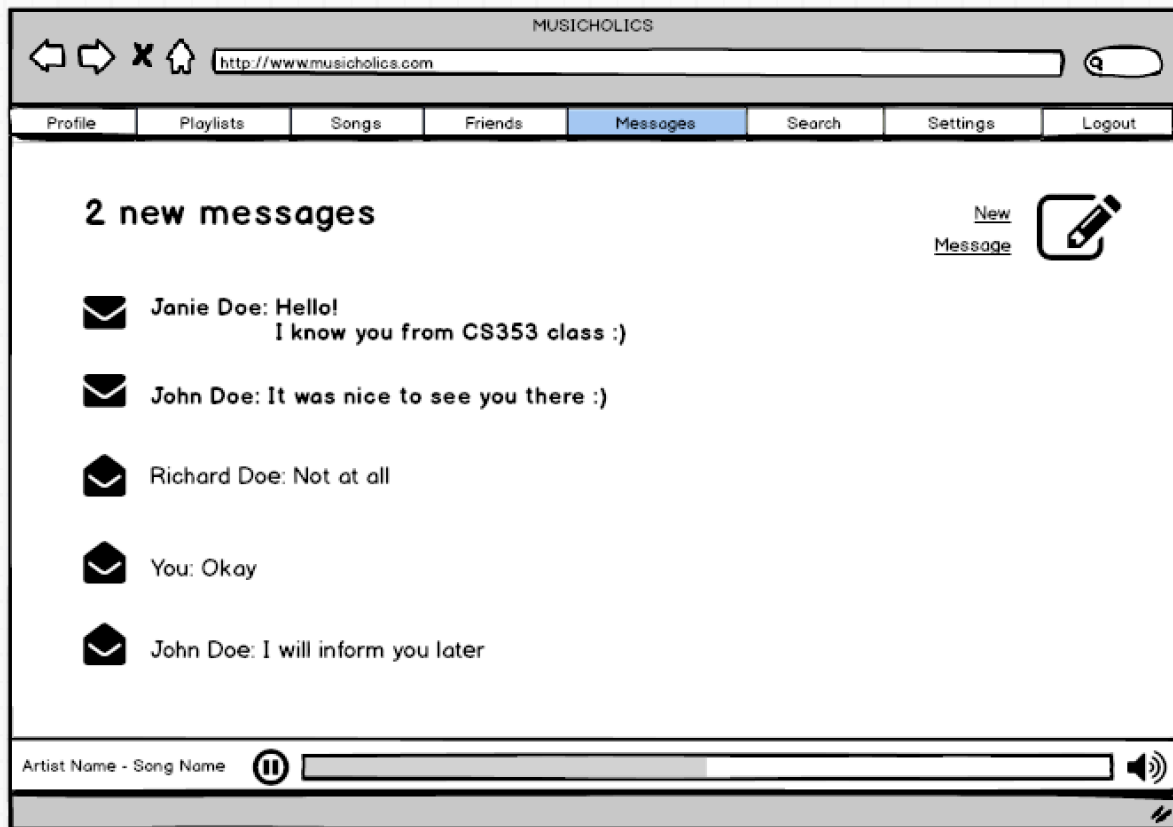
Process: The admin can search for a track, album, artist, playlist, user.

SQL Statement

No SQL Statement is needed because KMP algorithm will be used to search keyword in the data.

5.14.Messages

5.14.1. Message List



Input: @person_id

@person_id is id of the user logged in.

Process: The user can see the messages sent by other users.

SQL Statements

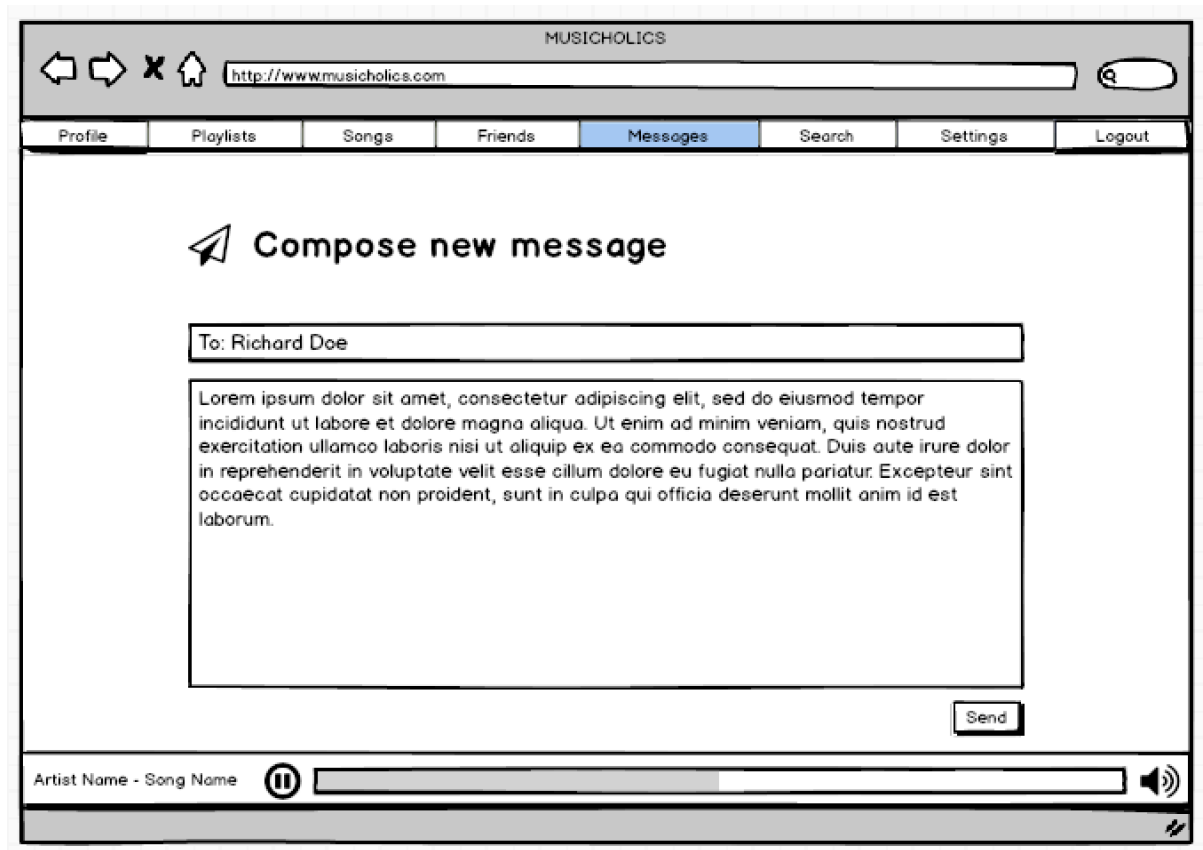
```
SELECT U.fullname, SM.message
```

```
FROM User U, Sends_Message SM
```

```
WHERE @person_id = SM.receiver_id AND U.id = SM.sender_id
```

```
ORDER BY SM.date
```

5.14.2.New Messages



Input: @person_id, @receiver_id, @message

@person_id is id of the message sender person.

@user_id is id of the message receiver person.

@message is the message that is sent.

Process: Logged in user, in other words message sender user, enters username of the receiver. Then full name of the receiver appears. Then, the user is able to send the message.

SQL Statements

Send Message:

```
INSERT INTO Sends_Message
```

```
VALUES(@person_id, @receiver_id, @GETDATE(), @message)
```

Receiver Information:

SELECT fullname

FROM User

WHERE @receiver_id = person_id

5.15.Purchase

5.15.1.Purchase Track with Credit Card

Name	Price
Song 1	\$3.76
Song 2	\$4.03
Song 3	\$2.23

Cardholder's Name

Credit Card Number

Expires on:

Card Security Code

Submit Cancel

Total: \$10.02

Artist Name - Song Name

Input: @person_id, @cardholder_name, @credit_card_number, @expire_month, @expire_year, @card_security_code, @membership_type

@person_id is id of the logged in user. @membership_type is membership type of the logged in user.

@cardholder_name, @credit_card_number, @expire_month, @expire_year, and, @card_security_code are entered by the user and not stored in database.

Process: The user enters his credit card information to buy the tracks. There is 50% discount for premium users. After purchase, tracks will be removed from card.

SQL Statements

Track Information:

```
SELECT T.track_name, T.price
```

```
FROM Track T, Buys B
```

```
WHERE B.user_id = @person_id AND T.track_id = B.track_id
```

Actual Price:

```
SELECT SUM(T.price) AS actual_price  
FROM Track T, Buys B  
WHERE B.user_id = @person_id AND T.track_id = B.track_id
```

Remove Tracks:

```
DELETE FROM Buys  
WHERE user_id = @person_id
```

Total Price:

```
SELECT total_price =  
    CASE @membership_type  
    WHEN "Premium" THEN ActualPrice / 2  
    ELSE ActualPrice  
END
```

5.15.2.Purchase Track with Budget

The screenshot shows a web browser window titled "MUSICHOLICS" with the URL "http://www.musicholics.com". The browser has navigation buttons (back, forward, stop, home) and a search bar. Below the browser is a navigation menu with links: Profile, Playlists, Songs, Friends, Messages, Search, Settings, and Logout. The main content area is titled "Purchase track using your budget". It contains a table with two columns: "Name" and "Price". The table lists three items: "Song 1" with price "\$3.76", "Song 2" with price "\$4.03", and "Song 3" with price "\$2.23". To the right of the table, the text "Total: \$10.02" is displayed. Below the table, the text "Your budget: \$32.57" is shown. At the bottom of this section are two buttons: "Purchase" and "Cancel". At the very bottom of the browser window is a music player bar showing "Artist Name - Song Name", a play/pause button, a progress bar, and a volume icon.

Name	Price
Song 1	\$3.76
Song 2	\$4.03
Song 3	\$2.23

Total: \$10.02

Your budget: \$32.57

Input: @person_id, @budget, @membership_type

@person_id is id of the logged in user. @membership_type is membership type of the logged in user. @budget is remaining money in the user's wallet.

Process: The user can use the money in his wallet to buy the tracks.

SQL Statements

Similar to 5.15.1.

Update budget:

UPDATE User

SET budget = budget – total_price

WHERE @person_id = person_id

5.15.3.Purchase Premium with Credit Card

The screenshot shows a web browser window with the URL <http://www.musicholics.com>. The browser's address bar and navigation buttons are visible. The website's navigation menu includes links for Profile, Playlists, Songs, Friends, Messages, Search, Settings, and Logout. The main content area features a large heading "Upgrade to premium" with a musical note icon. Below the heading, it states "Premium membership fee: \$9.99". The form contains the following fields: "Cardholder's Name", "Credit Card Number", "Expires on:" (with dropdowns for month and year, showing 01 and 2020), and "Card Security Code". There are "Submit" and "Cancel" buttons at the bottom of the form. At the very bottom of the browser window, a music player interface is visible, showing "Artist Name - Song Name", a play button, a progress bar, and a volume icon.

Input: @person_id, @cardholder_name, @credit_card_number, @expire_month, @expire_year, @card_security_code

@person_id is id of the logged in user.

@cardholder_name, @credit_card_number, @expire_month, @expire_year, and @card_security_code are entered by the user and not stored in database.

Process: The user enters his credit card information to get a premium account if the balance is enough

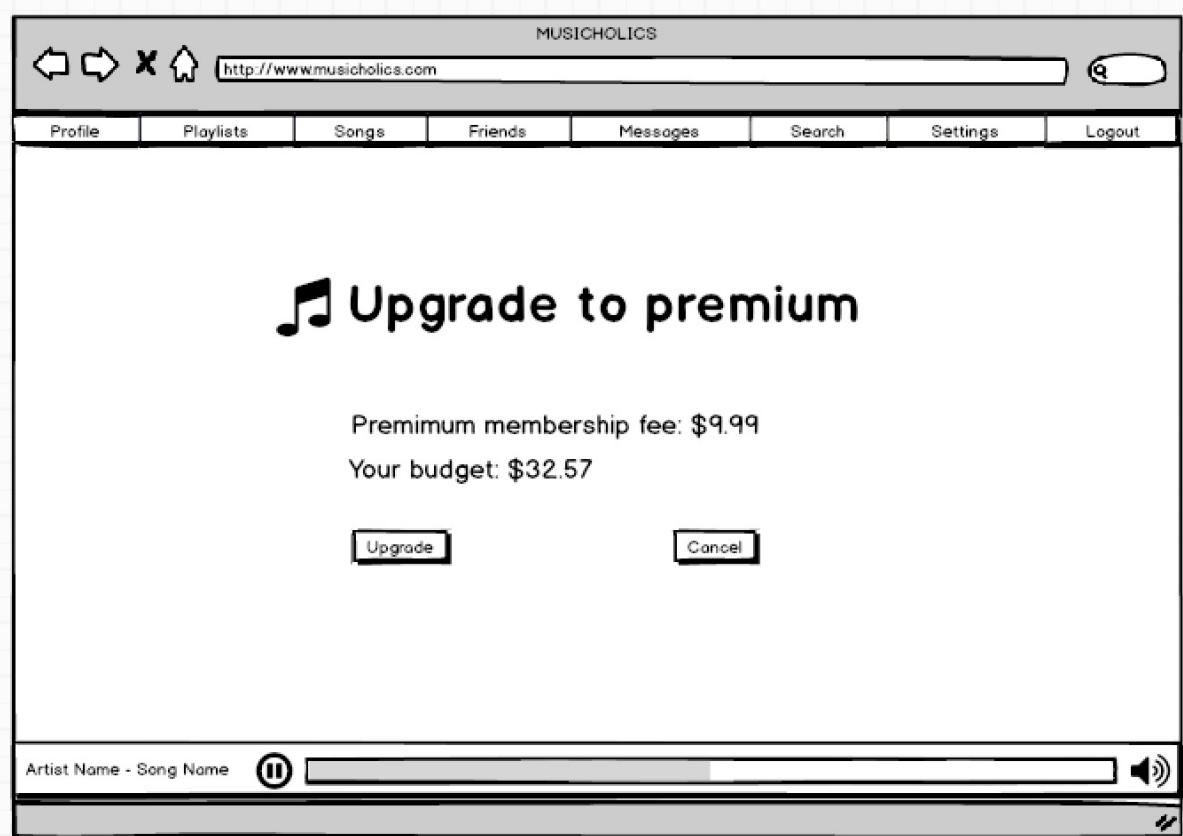
SQL Statements

UPDATE User

SET membership_type = "Premium"

WHERE person_id = @person_id

5.15.4. Purchase Premium with Budget



Input: @person_id, @budget

@person_id is id of the logged in user. @budget is remaining money in the user's wallet.

Process: The user can use the money in his wallet to get premium account if there is enough money.

SQL Statements

Similar to 5.15.3.

Update budget:

```
UPDATE User
```

```
SET budget = budget - 9.99
```

```
WHERE @person_id = person_id
```

5.15.5. Purchase Premium with Budget

The screenshot shows a web browser window with the URL <http://www.musicholics.com>. The browser's address bar and tabs are visible. The website's navigation bar includes links for Profile, Playlists, Songs, Friends, Messages, Search, Settings, and Logout. The main content area is titled "Add to your budget" and contains a form with the following fields: "Amount: \$" followed by an "Amount" input field, "Cardholder's Name", "Credit Card Number", "Expires on:" with dropdown menus for "01" and "2020", "Card Security Code", and "Submit" and "Cancel" buttons. At the bottom of the page, there is a music player interface showing "Artist Name - Song Name" and a progress bar.

Input: @person_id, @cardholder_name, @credit_card_number, @expire_month, @expire_year, @card_security_code, @amount

@person_id is id of the logged in user.

@cardholder_name, @credit_card_number, @expire_month, @expire_year, @amount, and @card_security_code are entered by the user and not stored in database.

Process: The user can add money to her wallet if she has enough balance.

SQL Statements

UPDATE User

SET budget = budget + @amount

WHERE @person_id = person_id

6. Advanced Data Components

6.1. Views

Users may not be allowed to view all attributes of entities. Therefore we will use views to restrict the view of some entities for some specific functions.

In this section @u_id is used as the id of the user who currently logged in.

6.1.1. View a Friend's Profile

A user is only able to see username, playlists, posts on his/her profile, email, country, language, birthday, gender and picture of another user. Additionally, he/she can view the song that is last listened by the viewed user. This will use when a user wants to display other user's profile.

```
CREATE VIEW view_friend AS
SELECT U.picture, U.username, U.fullname, U.email, U.country, U.language,
U.birthday, U.gender, PL.playlist_name, P.post, P.date, T.track_name
FROM User U, Playlist PL, Post P, Track T, Listens L
WHERE U.user_id = @us_id AND U.user_id = PL.creator_id
      AND P.receiver_id= U.user_id
      AND L.user_id = U.user_id
      AND T.track_id = ( SELECT L1.track_id
                        FROM Listens L1
                        WHERE L1.user_id = U.user_id
                        AND L1.date = ( SELECT max( L2.date)
                                      FROM Listens L2
                                      WHERE L2.user_id = U.user_id ))
```

@us_id is the id of the user that is to be viewed.

6.1.2. View a Profile of a User Not a Friend

```
CREATE VIEW view_unfriend AS
SELECT U.picture, U.username, U.fullname, U.country
FROM User U
WHERE U.user_id = @us_id
```

@us_id is the id of the user that is wanted to be viewed.

6.1.3. Blocked Users

If a user is blocked, s/he cannot see the details of profile of the user who banned him/her.

Blocker and blocked users can see only usernames of each others.

```
CREATE VIEW blocked_user AS
SELECT U2.username
FROM User U1, User U2, Blocks B
WHERE( U1.user_id = B.blocker_id AND U1.user_id = @u_id
      AND U2.user_id = B.blocked_id )
OR
( U1.user_id = B.blocked_id AND U1.user_id = @u_id
  AND U2.user_id = B.blocker_id)
AND U1.user_id = @u_id
```

6.1.4. Message View

A person can only view messages that s/he sent or received.

```
CREATE VIEW message_view AS
SELECT *
FROM Sends_Message
WHERE @u_id = sender_id OR @u_id = receiver_id
```

6.1.5. Album view

A user is able to view name, artist's name, song's name, song's duration, song's price of a album.

```
CREATE VIEW album_view AS
SELECT T.track_name, T.duration, T.price, AR.artist_name, AL.album_name
FROM Album AL, Artist AR, Track T, Track_Belongs_To_Album ATA,
      Album_Belongs_To_Artist ART
WHERE AL.album_id = @a_id AND T.track_id = ATA.track_id
      AND ATA.album_id = AL.album_id AND ART.artist_id = AR.artist_id
```

@a_id is the id of the album that is to be viewed.

6.2. Stored Procedures

By using stored procedures, the design of the database system of the application will be simplified. These can be used rather than writing queries which are used frequently. They can be considered as functions in high level languages, so they ease the programmer job. Required stored procedures can be found in below.

- We will show the recent activity of a user's friends. Therefore when a user enters to the application, the recent listened song of his/her friends must be shown on the screen.
- The budget of a user must be calculated each time that s/he buys a track. Then it must be displayed to the user.
- When a user buy a track as a gift for another user, gift list of both users must be updated.
- When a user sends a message to another user, message boxes of each user must be updated immediately.
- When a user rates a playlist, average rate point of the playlist must be updated and displayed.
- When a user comments on playlist, comments of the playlist must be updated and displayed.
- When a user follows a playlist, follower number of the playlist must be updated and displayed.
- When admin bans a user, this user's information must be deleted and the account must be removed from the database.
- When a user blocks a user and if they are friend, the corresponding tuple for their friendship must be removed.

6.3. Reports

Reports generally will be used counting an information related to the entities. This will ease the system by hindering to write queries very frequently.

6.3.1. Total number of tracks in a playlist

```
SELECT COUNT(*)  
FROM Added A, Playlist P  
WHERE A.playlist_id = P.playlist_id
```

6.3.2. Total number of followers of a playlist

```
SELECT COUNT(*)  
FROM Follows F, Playlist P  
WHERE F.playlist_id = P.playlist_id
```

6.3.3. Total number of comments of a playlist

```
SELECT COUNT(*)  
FROM Comments C, Playlist P  
WHERE C.playlist_id = P.playlist_id
```

6.3.4. Average rate of a playlist

```
SELECT AVG(R.rate)  
FROM Rates R, Playlist P  
WHERE R.playlist_id = P.playlist_id
```

6.3.5. Total number of playlist owned by a user

```
SELECT COUNT(*)  
FROM User U, Playlist P  
WHERE U.user_id = P.user_id
```

6.3.6. Total duration of a playlist

```
SELECT SUM(T.duration)
FROM Added A, Playlist P, Track T
WHERE A.playlist_id = P.playlist_id AND A.track_id = T.track_id
```

6.3.7. Total number of tracks of a album

```
SELECT COUNT(*)
FROM Album
GROUP BY album_id
```

6.3.8. Total number of user

```
SELECT COUNT(*)
FROM USER
```

6.3.9. Total number of tracks that is purchased

```
SELECT COUNT(*)
FROM Buys
GROUP BY user_id
```

6.3.10. Total number of tracks that is listened by a user

```
SELECT COUNT(*)
FROM Listens
GROUP BY user_id
```

6.3.11. Total number of collaborator of a playlist

```
SELECT COUNT(*)
FROM Collaborates
GROUP BY playlist_id
```

6.3.12. Total number of friends of a user

```
SELECT COUNT(*)  
FROM Friendship  
WHERE user1_id
```

6.3.13. Total number of chats of a user

```
SELECT COUNT(sender_id)  
FROM Sends_Message  
WHERE receiver_id
```

6.4. Triggers

- When a playlist deleted; rates of it, comments of it, followers relation, and belonging relation with its tracks must be deleted.
- When a user is deleted; messages, playlists, comments and rating related to the user must be deleted. Playlist's average rate point must be updated.
- When a user blocks the other, friendship relation must be deleted between the users, if any.
- When a publisher deleted from the database, albums owned by it must be deleted.
- When an artists deleted from the database, tracks possessed by him/her must be deleted.
- If a album is owned by a single artist, when a artist is deleted the album must be deleted also.
- When an album is deleted, tracks of that album must be deleted.
- When a track is deleted, gift, listening activity, and buying activity must be removed. Also it must be removed from the playlist that it is found.

6.5. Constraints

- Users must have to create an account and then log i the system in order to listen music.
- Users must purchase tracks in order to listen and add them to their playlists.
- Users must be friend to see their listening activities.
- Users must be friend to purchase and send a track as a gift.
- A playlist must be belong to a single user.
- A track must be belong to an album.
- An album must be published by a single publisher.
- A track must be belong to an artist.
- An album can be belong to several artist and a artist can have several albums.
- A artist can have at most 150 albums.
- An album can be belong to at most 10 artist.
- A playlist can have at most 500 tracks.
- A user can create at most 500 playlists.
- A user can write a comment with at most 300 characters.
- A user can rate a playlist as at most 5, at least 0.
- A user can follow at most 800 users.
- A username can consists of at most 15, at least 5 characters.
- A password can consists of at most 20, at least 6 characters.
- An e-mail address must be valid.
- A birthday year can be at most 2010, at least 1930.
- A message can have at most 1500 characters.
- Price of a track must be more than 0.

7. Implementation Plan

In order to meet the functional requirements of the system, we will use PHP, HTML, CSS, JavaScript, and Bootstrap for front-end. Java will be used to implement back-end of the application. We will use MySQL for the database system of the application. MySQL Connector/J library will be used to connect back-end and database of the system.

8. Online Access

The website link of the application can be found below.

<https://github.com/miracvbasaran/Musicocholics>