Bilkent University

Department of Computer Engineering

CS 353 - Database Systems

HeyListen: Music Playing Database System

Project Design Report

Group 16

Arkın Yılmaz	21502080	Section 2
Anıl Erken	21501468	Section 2
Berk Mandıracıoğlu	21501741	Section 1
İrem Ural	21502278	Section 2

Course Instructor: Özgür Ulusoy

April 2, 2018

1.0 REVISED E/R MODEL

According to the given feedback and during our design decisions we changed or modified the following parts of the E/R diagram:

1.We removed some of the attributes, as they can be dynamically computed from relations.These attributes are:

- number of followers and following in user entity
- number of songs and followers of the playlist
- number of sold_songs and number_of_sold_albums of the singer
- number of likes in song

2. We changed the relation between user and comment and made comment an entity rather than a weak entity as we can define a primary key for the comment.

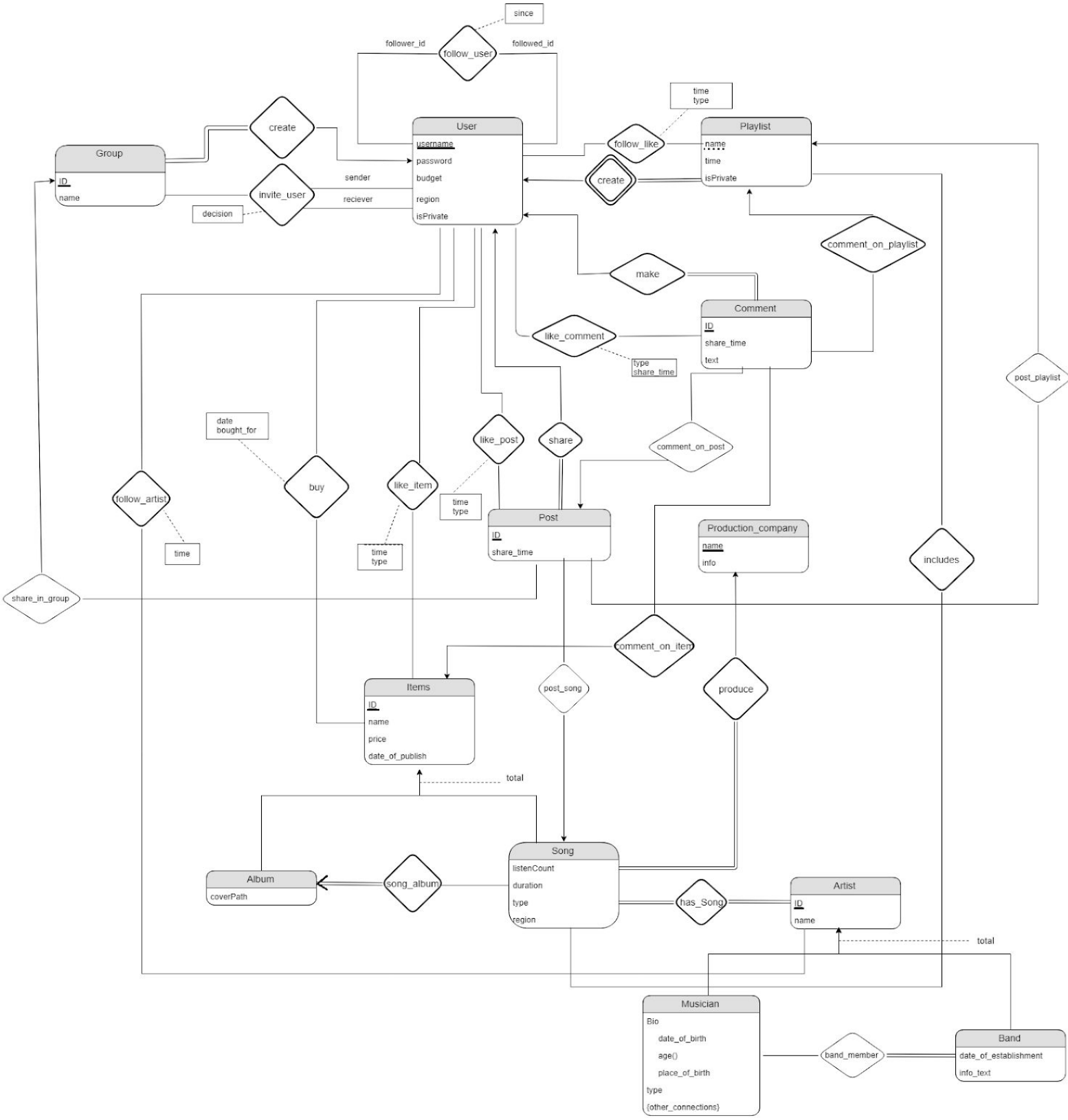
3. We added a create relation between group and user, where group side is total and M, user side is 1.Additionally, we made the invite relation ternary and added roles as sender and receiver for users.

4. We removed the purchase_ID attribute of the relation ‘buy’.

5. We changed the share relation and we created an entity as post, which has a unique ID and a share_time. Then a new relation called share between post and user is created, which is 1-M(many in post side) and post has a total participation. Then between post and song post_song relation is defined, as well as post_playlist relation between post and playlist. Lastly, to identify the posts in the group we added a relation between group and post called share_in_group.These last three relations are 1-M (many in post side).

6. We thought that we need to have a band entity as well in our database. We created a new entity called artist and by using specialization we link it to band and musician(which was defined as singer entity before). ID and name attributes, which are common to band and musician are located in artist. Then between musician and band a new relation band_member is defined, which shows the musicians who has a band.

7. We removed the ID of production company and make its name as the primary key.
8. We named some of the attributes as time rather than date.


since
follower_id
follow_user
followed_id
time
type
create
User
username
password
budget
region
isPrivate
follow_like
Playlist
name
time
isPrivate
Group
ID
name
invite_user
sender
reciever
decision
create
comment_on_playlist
make
Comment
ID
share_time
text
like_comment
type
share_time
post_playlist
date
bought_for
like_post
share
comment_on_post
buy
like_item
time
type
follow_artist
time
time
type
Post
ID
share_time
Production_company
name
info
includes
share_in_group
comment_on_item
Items
ID
name
price
date_of_publish
post_song
produce
total
Song
listenCount
duration
type
region
Album
coverPath
song_album
has_Song
Artist
ID
name
total
Musician
Bio
date_of_birth
age()
place_of_birth
type
{other_connections}
band_member
Band
date_of_establishment
info_text

2.0 RELATION SCHEMAS

2.1 USER

Relational Model:

user (<u>username</u>, password, budget, region, isPrivate)

Functional Dependencies:

username -> password, budget, region, isPrivate

Candidate Keys:

{username}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE user(
        username VARCHAR(45)  PRIMARY KEY,
        password VARCHAR(25)   NOT NULL,
        budget     TINYINT NOT  NULL,
        region     VARCHAR(30)  NOT NULL,
        isPrivate  BIT(1)              NOT NULL
);
```

2.2 PRODUCTION COMPANY

Relational Model:

production_company(name, info)

Functional Dependencies:
name -> info

Candidate Keys:
{name}

Normal Form:
BCNF

Table Definition:

```
CREATE TABLE production_company(
        name  VARCHAR(45)  PRIMARY KEY,
        info VARCHAR(256)
);
```

2.3 PLAYLIST

Relational Model:

playlist (<u>username, name</u>, time, isPrivate)
FK: username references user(username)

Functional Dependencies:
username, name -> time, isPrivate

Candidate Keys:
{ (username, name) }

Normal Form:
BCNF

Table Definition:

```
CREATE TABLE playlist(
        username      VARCHAR(45) ,
        name          VARCHAR(45)  NOT NULL,
time            TIMESTAMP DEFAULT CURRENT TIMESTAMP,
        isPrivate     BIT(1) NOT NULL,
        PRIMARY KEY ( (username, name) )
);
```

2.4 FOLLOW_ARTIST

Relational Model:
follow_artist (<u>username, ID,</u> time)

FK: username references user(username)
FK: ID references artist(ID)

Functional Dependencies:
username, ID -> time

Candidate Keys:
{(username, ID)}

Normal Form:
BCNF

Table Definition:

```
CREATE TABLE follow_artist(
        username      VARCHAR(45) ,
        ID                INT,
        PRIMARY KEY ( (username, ID) )
        FOREIGN KEY (ID) REFERENCES artist(ID),
        FOREIGN KEY (username) REFERENCES user(username)
)
ENGINE = InnoDB;
```

2.5 PLAYLIST_INCLUDES

Relational Model:
playlist_includes(<u>username,name, songID</u>)
FK: (username,name) references playlist(username,name)
FK: songID references song(ID)

Functional Dependencies:
None

Candidate Keys:
{(username, name, songID**)}**

Normal Form:
BCNF

Table Definition:

```
CREATE TABLE playlist_includes(
        username      VARCHAR(45) ,
        name          VARCHAR(45),
        songID        INT,
        PRIMARY KEY ( (username, ID, songID) )
        FOREIGN KEY (songID) REFERENCES song(ID),
        FOREIGN KEY (username,name) REFERENCES playlist(username,name)
)
ENGINE = InnoDB;
```

2.6 FOLLOW_LIKE_PLAYLIST

Relational Model:
follow_like_playlist (<u>userUsername</u>, <u>creatorUsername</u>, <u>creatorName</u>, <u>type</u>, time)
FK: userUsername references user(username)
FK: (creatorUsername, creatorName) references playlist(username,name)

Functional Dependencies:
FD: userUsername, creatorUsername, creatorName, type -> time

Candidate Keys:
{ (userUsername, creatorUsername, creatorName, type)**}**

Form:
BCNF

Table Definition:

```
CREATE TABLE follow_like_playlist(
        userUsername        VARCHAR(45) ,
        creatorUsername     VARCHAR(45),
        creatorName         INT,
        type                TINYINT,
        time_               TIMESTAMP DEFAULT CURRENT  TIMESTAMP,
        PRIMARY KEY ( (userUsername, creatorUsername,creatorName, type) ),
        FOREIGN KEY (userUsername) REFERENCES user(username),
FOREIGN KEY (creatorUsername,creatorName) REFERENCES
        playlist(username,name)
)
ENGINE = InnoDB;
```

2.7 FOLLOW_USER

Relational Model:
follow_user (follower_id,followed _id, since)
FK: follower_id references user(username)
FK: follow_id references user(username)

Functional Dependencies:
follower_id, followed_id -> since

Candidate Keys:

{(follower_id, followed_id)}

Form:
BCNF

Table Definition:

```
CREATE TABLE follow_user(
        follower_id      VARCHAR(45) ,
        followed_id      VARCHAR(45),
since              DATETIME DEFAULT 0 ON UPDATE CURRENT
   TIMESTAMP,
        PRIMARY KEY ( (followed_id, follower_id) )
FOREIGN KEY (followed_id) REFERENCES (username)
FOREIGN KEY (follower_id) REFERENCES (username)
)
ENGINE = InnoDB;
```

2.8 GROUP

Relational Model:
group(ID, admin, name)
FK: admin references user(username)

Functional Dependencies:
ID -> admin, name

Candidate Keys:
{(ID)}

Form:
BCNF

Table Definition:

```
CREATE TABLE group(
        ID              INT PRIMARY KEY AUTO_INCREMENT,
        admin           VARCHAR(45),
name            VARCHAR(45),
FOREIGN KEY (admin) REFERENCES user(username)
)
ENGINE = InnoDB;
```

2.9 INVITE_USER

Relational Model:
invite_user(groupID, sender, reciever, decision)
FK: groupID references group(ID)
FK: receiver references user(username)
FK: sender references user(username)

Functional Dependencies:
groupID, sender, reciever -> decision
groupID -> sender
groupID, receiver - > sender
sender, receiver -> decision

groupID -> sender violates BNCF as it is not a superkey or trivial.

In BCNF FORM there are two relations. These are the following:

1. group(groupID, sender)

This relation exists above as group. Sender is defined as admin.

Functional Dependency:
groupID ->groupID, sender

2. group_invite(groupID, reciever, decision)

Functional Dependency:
groupID, reciever -> decision
Hence,

invite_user (groupID, reciever, decision)
FK: groupID references group(ID)

FK: receiver references user(username)

Candidate Keys:
{(groupID, reciever)}

Form:
BCNF

Table Definition:

```
CREATE TABLE invite_user(
        groupID             INT,
        reciever            VARCHAR(45),
decision            BIT(2),
        PRIMARY KEY ( (groupID,receiver) )
FOREIGN KEY (reciever) REFERENCES user(username)
)
ENGINE = InnoDB;
```

2.10 COMMENT

Relational Model:
comment (ID, username, share_time, text)
FK: username references user(username)

Functional Dependencies:
ID -> username, username, share_time, text

Candidate Keys:
{(ID)}

Form:
BCNF

Table Definition:

```
CREATE TABLE invite_user(
        ID                  INT PRIMARY KEY AUTO_INCREMENT,
        username            VARCHAR(45),
share_time          TIMESTAMP DEFAULT CURRENT TIMESTAMP,
        text                VARCHAR(256),
```

```
FOREIGN KEY (username) REFERENCES user(username)
)
ENGINE = InnoDB;
```

2.11 LIKE_COMMENT

Relational Model:

like_comment(liker , ID, share_time, type)
FK: liker references user(username)
FK: ID references comment(ID)

Functional Dependency:

liker, ID-> share_time, type

Candidate Keys:

{(liker, ID)}

Form:

BCNF

Table Definition:

```
CREATE TABLE like_comment(
        liker                   VARCHAR(45) NOT NULL,
        ID                      INT  NOT NULL,
share_time          TIMESTAMP DEFAULT CURRENT TIMESTAMP,
        text                    VARCHAR(256),
        type                    BIT(1),
        PRIMARY KEY ( liker, ID )
FOREIGN KEY (liker) REFERENCES user(username)
FOREIGN KEY (ID) REFERENCES comment(ID)
)
ENGINE = InnoDB;
```

2.12 POST

Relational Model:
post (ID, share_time, username)
FK: username references user(username)

Functional Dependencies:
ID -> share_time, username

Candidate Keys:
{(ID)}

Form:
BCNF

Table Definition:

```
CREATE TABLE post(
        ID                      INT PRIMARY KEY AUTO_INCREMENT,
        username                VARCHAR(45) NOT NULL,
share_time          TIMESTAMP DEFAULT CURRENT TIMESTAMP,
FOREIGN KEY (username) REFERENCES user(username)
)
ENGINE = InnoDB;
```

2.13 POST_SONG

Relational Model:
post_song(p_ID, s_ID)
FK: s_ID references song(ID)
FK: p_ID references post(ID)

Functional Dependencies:
None

Candidate Keys:
{(p_ID, s_ID)}

Form:
BCNF

Table Definition:

```
CREATE TABLE post_song(
        p_ID                    INT NOT NULL,
        s_ID                    INT NOT NULL,
        PRIMARY KEY ( s_ID, p_ID )
FOREIGN KEY (p_ID) REFERENCES post(ID)
FOREIGN KEY (s_ID) REFERENCES song(ID)
)
ENGINE = InnoDB;
```

2.14 POST_PLAYLIST

Relational Model:
post_playlist(post_ID, p_name, p_username)
FK: (p_name, p_username) references playlist (name, username)
FK: post_ID references post(ID)

Functional Dependencies:
None

Candidate Keys:
{ (post_ID, p_name, p_username)}

Form:
BCNF

Table Definition:

```
CREATE TABLE post_playlist(
        post_ID                 INT NOT NULL,
        p_name                  VARCHAR(45),
        p_username              VARCHAR(45),
        PRIMARY KEY ( (post_ID, p_name, p_username) )
FOREIGN KEY (post_ID) REFERENCES post(ID)
FOREIGN KEY (p_name, p_username) REFERENCES playlist (name, username)
)
```

```
ENGINE = InnoDB;
```

2.15 SHARE_IN_GROUP

Relational Model:
share_in_group(post_ID, g_ID)
FK: g_ID references group(ID)
FK: post_ID references post(ID)

Functional Dependencies:
post_ID, g_ID -> post_ID, g_ID

Candidate Keys:
{(postID, g_ID)}

Form:
BCNF

Table Definition:

```
CREATE TABLE share_in_group(
        post_ID                 INT NOT NULL,
        g_ID                    INT NOT NULL,
        PRIMARY KEY ( (post_ID, g_ID) )
FOREIGN KEY (post_ID) REFERENCES post(ID)
FOREIGN KEY (g_ID) REFERENCES group (ID)
)
```

```
ENGINE = InnoDB;
```

2.16 LIKE_POST

Relational Model:
like_post(u_username, p_ID, time, type)
FK: u_username references user(username)
FK: p_ID references post(ID)

Functional Dependencies:
u_username, p_ID -> time, type

Candidate Keys:
{(u_username, p_ID)}

Form:
BCNF

Table Definition:

```
CREATE TABLE like_post(
        u_username          VARCHAR(45),
        p_ID                INT NOT NULL,
        time                TIMESTAMP DEFAULT CURRENT TIMESTAMP,
        type                BIT(1),
```

```
        PRIMARY KEY ( (u_username, p_ID) )
FOREIGN KEY (p_ID) REFERENCES post(ID)
FOREIGN KEY (u_username ) REFERENCES user(username)
)
ENGINE = InnoDB;
```

2.17 LIKE_ITEM

Relational Model:
like_item(<u>username, itemID</u>, time, type)
FK: username references user(username)
FK: itemID references items(ID)

Functional Dependencies:
username, itemID -> time, type

Candidate Keys:
{(username, itemID)}

Form:
BCNF

Table Definition:

```
CREATE TABLE like_item(
        username            VARCHAR(45),
        itemID              INT NOT NULL,
        time                TIMESTAMP DEFAULT CURRENT TIMESTAMP,
```

```
        type                    BIT(1),
        PRIMARY KEY ((username, itemID) )
FOREIGN KEY (itemID) REFERENCES items(ID)
FOREIGN KEY (username ) REFERENCES user(username)
)
ENGINE = InnoDB;
```

2.18 BUY

Relational Model:
buy(buyer, itemID, bought_for, date)
FK: buyer references user(username)
FK: bought_for references user(username)
FK: itemID references items(ID)

Functional Dependencies:
buyer, itemID, bought_for -> date

Candidate Keys:
{(buyer, itemID, bought_for)}

Form:
BCNF

Table Definition:

```
CREATE TABLE buy(
        buyer                   VARCHAR(45),
```

```
        itemID                  INT NOT NULL,
        bought_for              VARCHAR(45)
        date                    DATETIME DEFAULT 0 ON UPDATE CURRENT
          TIMESTAMP,
        PRIMARY KEY (buyer, itemID, bought_for)
FOREIGN KEY (itemID) REFERENCES items(ID)
FOREIGN KEY (buyer) REFERENCES user(username)
FOREIGN KEY (bought_for ) REFERENCES user(username)
)
ENGINE = InnoDB;
```

2.19 ITEMS

Relational Model:
items(ID, name, price, date_of_publish)

Functional Dependencies:
ID -> name, price, date_of_publish

Candidate Keys:
{(ID)}

Form:
BCNF

Table Definition:

```
CREATE TABLE items(
        ID                      INT PRIMARY KEY AUTO_INCREMENT
        name                    VARCHAR(45),
        price                   TINYINT
```

```
        date_of_publish       DATETIME DEFAULT 0 ON UPDATE CURRENT
         TIMESTAMP
);
```

2.20 ALBUM

Relational Model:
album(ID, coverpath)
FK: ID references items(ID)

Functional Dependencies:
ID -> coverpath

Candidate Keys:
{(ID)}
Form:
BCNF

Table Definition:

```
CREATE TABLE album(
        ID                      INT NOT NULL,
        coverPath               VARCHAR(256),
        PRIMARY KEY (ID)
```

);

2.21 ARTIST

Relational Model:
artist(ID, name)

Functional Dependencies:
ID -> name

Candidate Keys:
{(ID)}
Form:
BCNF

Table Definition:

```
CREATE TABLE artist(
        ID                      INT PRIMARY KEY AUTO_INCREMENT
        name                    VARCHAR(45),
);
```

2.23 SONG

Relational Model:
song(ID, duration, listenCount, type, region, companyName , albumID)
FK: ID references items(ID)
FK: companyName references production_company (name)
FK: albumID references album(ID)

Functional Dependencies:
ID -> duration, listenCount, type, region, companyName , albumID

Candidate Keys:
{(ID)}
Form:
BCNF

Table Definition:

```
CREATE TABLE song(
```

```
        ID                      INT NOT NULL,
        duration                INT,
        listenCount             INT,
        type                    VARCHAR(45),
        region                  VARCHAR(45) NOT NULL,
        companyName             VARCHAR(45) NOT NULL,
        albumID                 INT,
        PRIMARY KEY (ID),
FOREIGN KEY (ID) REFERENCES items(ID),
FOREIGN KEY (companyName) REFERENCES production_company (name)
FOREIGN KEY (albumID  ) REFERENCES  album(ID)
)
ENGINE = InnoDB;
```

2.24 MUSICIAN

Relational Model:
musician(ID, date_of_birth, place_of_birth, type)
FK: ID references artist(ID)

Functional Dependencies:
ID -> date_of_birth, place_of_birth

Candidate Keys:
{(ID)}
Form:
BCNF

Table Definition:

```
CREATE TABLE musician(
        ID                      INT NOT NULL,
        date_of_birth           DATETIME ,
```

```
        place_of_birth        VARCHAR(45),
        type                  VARCHAR(45),
        PRIMARY KEY (ID),
FOREIGN KEY (ID) REFERENCES artist(ID),
)
ENGINE = InnoDB;
```

2.25 BAND_MEMBER

Relational Model:
band_member(musicianID, bandID)
FK: bandID references band(ID)
FK: musicianID references musician(ID)

Functional Dependencies:
None

Candidate Keys:
{(musicianID, bandID)}

Form:
BCNF

Table Definition:

```
CREATE TABLE band_member(
        musicianID        INT NOT NULL,
        bandID            INT NOT NULL,
        PRIMARY KEY ((musicianID, bandID)),
FOREIGN KEY (bandID) REFERENCES band(ID),
FOREIGN KEY musicianID REFERENCES musician(ID)
)
ENGINE = InnoDB;
```

2.26 BAND

Relational Model:
band(ID, date_of_establishment, info_text)
FK: ID references artist(ID)

Functional Dependencies:
ID -> date_of_establishment, info_text

Candidate Keys:
{(ID)}
Form:
BCNF

Table Definition:

```
CREATE TABLE band(
```

```
        ID                                  INT NOT NULL,
        date_of_establishment               DATETIME,
        info_text                           VARCHAR(256),
        PRIMARY KEY ((ID)),
FOREIGN KEY (ID) REFERENCES artist(ID),
)
ENGINE = InnoDB;
```

2.27 MUSICIAN_CONNECTIONS

Relational Model:
musician_connections(ID, other_connections)
FK: ID references musician(ID)

Functional Dependencies:
None

Candidate Keys:
{(ID, other_connections)}
Form:
BCNF

Table Definition:

```
CREATE TABLE band_member(
        ID                                  INT NOT NULL,
        other_connections                   VARCHAR(128),
        PRIMARY KEY ((ID)),
FOREIGN KEY (ID) REFERENCES musician(ID),
)
ENGINE = InnoDB;
```

2.28 HAS_SONG

Relational Model:

has_Song(artistID, songID)
FK: songID references song(ID)
FK: artistID references artist(ID)

Functional Dependencies:
None

Candidate Keys:
{(artistID, songID)}

Form:

BCNF

Table Definition:

```
CREATE TABLE has_Song(
        artistID                INT NOT NULL,
        songID                  INT NOT NULL,
        PRIMARY KEY ( (artistID, songID) ),
FOREIGN KEY (artistID) REFERENCES artist(ID),
FOREIGN KEY (songID) REFERENCES song(ID),
)
ENGINE = InnoDB;
```

Though these three below relations are 1-M and can be combined with comment entity, as the many side is not total, it causes a lot of null values. Therefore they are defined as separate tables in the database as the following:

2.29 COMMENT_ON_ITEM

Relational Model:
comment_on_item(<u>commentID,</u> itemID)
FK: commentID references comment(ID)
FK: itemID references items(ID)

Functional Dependencies:
commentID -> itemID

Candidate Keys:
{(commentID)}

Form:
BCNF

Table Definition:

```
CREATE TABLE comment_on_item(
        commentID                   INT NOT NULL,
        itemID                      INT NOT NULL,
        PRIMARY KEY ( (commentID) ),
FOREIGN KEY (commentID) REFERENCES comment(ID),
FOREIGN KEY (itemID) REFERENCES items(ID),
)
ENGINE = InnoDB;
```

2.30 COMMENT_ON_POST

Relational Model:
comment_on_post(<u>commentID,</u> postID)
FK: commentID references comment(ID)
FK: postID references post(ID)

Functional Dependencies:
commentID -> postID

Candidate Keys:
{(commentID)}

Form:

BCNF

Table Definition:

```
CREATE TABLE comment_on_post(
        commentID          INT NOT NULL,
        postID             INT NOT NULL,
        PRIMARY KEY ( commentID ),
FOREIGN KEY (commentID) REFERENCES comment(ID),
FOREIGN KEY (postID) REFERENCES post(ID),
)
ENGINE = InnoDB;
```

2.31 COMMENT_ON_PLAYLIST

Relational Model:

comment_on_playlist(<u>commentID</u>, owner, playlistName)
FK: commentID references comment(ID)
FK: (owner, playlistName) references playlist(username, name)

Functional Dependencies:
commentID -> owner, playlistName

Candidate Keys:
{(commentID)}

Form:
BCNF

Table Definition:

```
CREATE TABLE comment_on_playlist(
        commentID          INT NOT NULL,
        owner              INT NOT NULL,
        playlistName       VARCHAR(45) NOT NULL,
        PRIMARY KEY ( commentID ),
FOREIGN KEY (commentID) REFERENCES comment(ID),
FOREIGN KEY (owner, playlistName) REFERENCES playlist(username,
    name)
)
ENGINE = InnoDB;
```

3.0 FUNCTIONAL COMPONENTS

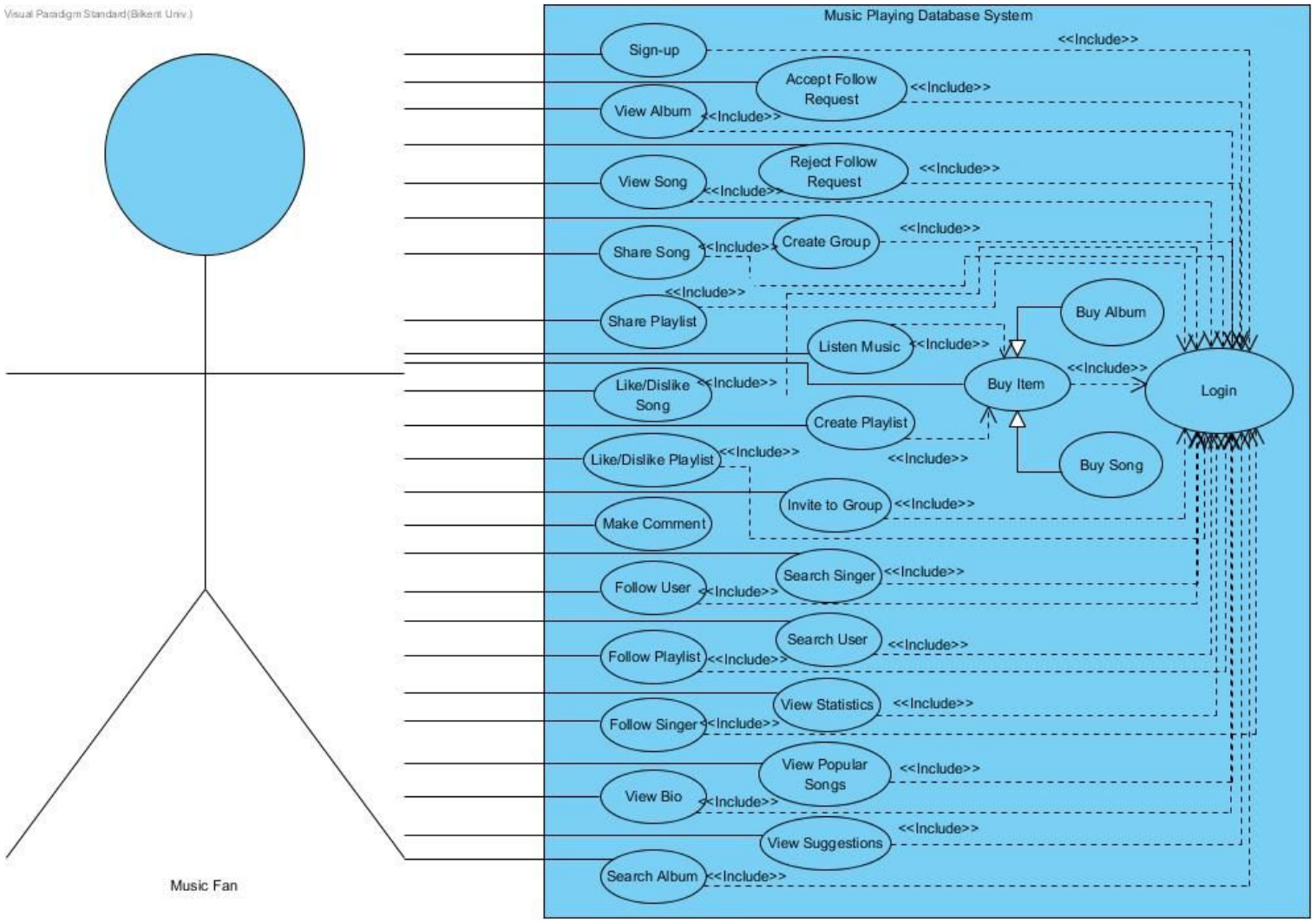
3.1 USE CASES AND SCENARIOS

Our Music Playing Database System is made for two types of end users: Music Fans and Music Producers. There are many functionalities that will satisfy all music fans with different interests. Also, the music producers can make use of the data provided by our application. These functional requirements are listed below:

3.1.1) Music Fan

- **Login:** User should login to system with a username and password, for all functionalities listed below.
- **Sign-up:** User should sign-up to system by registering his/her username and password.
- **View Album:** User should be able to see all albums of a singer in the database.
- **View Song:** User should be able to see all songs of a singer in the database. By navigatin in that menu, user can also see all songs in a particular album.
- **Buy Item:** User should be able to buy albums and songs.
- **Buy Album:** User should be able to buy the albums. User should also be able to buy an album for another user.
- **Buy Song:** User should be able to buy the songs. User should also be able to buy a song for another user.
- **Listen Music:** User should be able to listen all music in the application after buying them.
- **Create Playlist:** User should be able to create playlists. In order to add songs to playlist, user should buy the songs or albums.
- **Share Song:** User should be able to share songs.
- **Share Playlist:** User should be able to share playlists.
- **Like/Dislike Song**: User should be able to like or dislike shared songs.
- **Like/Dislike Playlist:** User should be able to like or dislike shared playlists.
- **Make Comment:** User should be able to make comments on shared playlists and songs.
- **Follow User:** User should be able to follow other users. In this way, user will see shared playlists and songs from the followed users.
- **Follow Playlist:** User should be able to follow playlists. In this way, user will see update in a followed playlists.
- **Follow Singer:** User should be able to follow singers. In this way, user will see the new songs from the followed singers.
- **Accept Follow Request:** User should be able to accept any incoming follow requests.
- **Reject Follow Request:** User should be able to reject any incoming follow requests.
- **Create Group:** User should be able to create groups. A group should have at least one admin and the creator of the group is admin by default.
- **Invite to Group:** User who is admin of the group should be able to invite other users to join to the group.

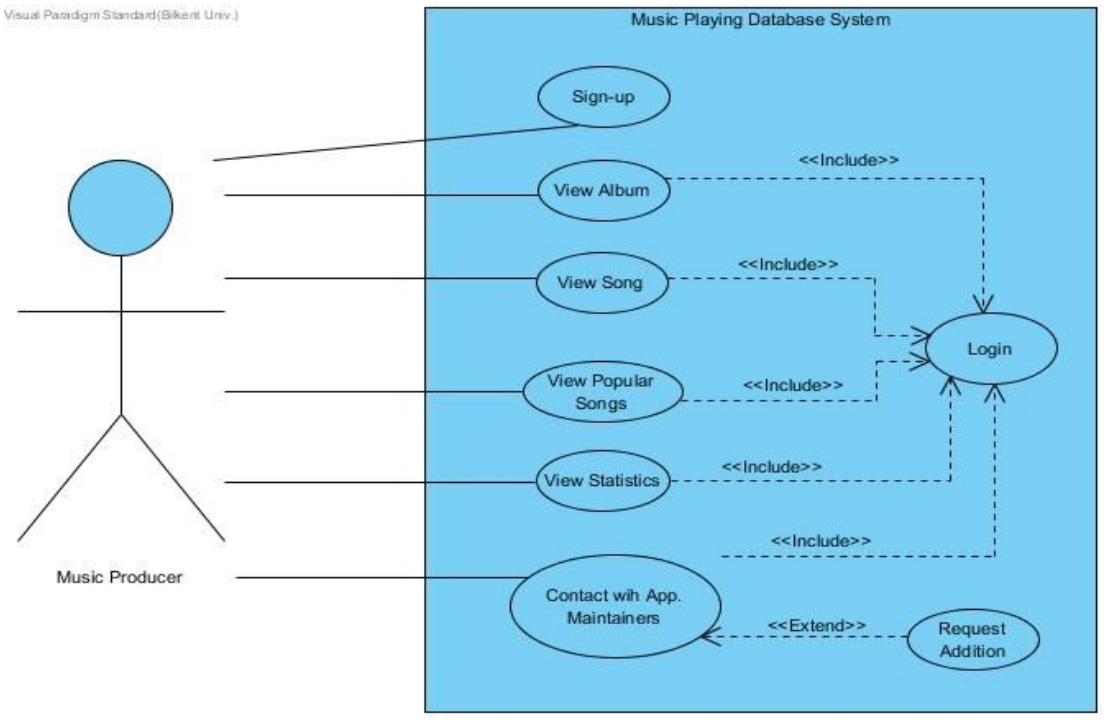
- **Join Group:** User should be able to join groups by accepting invitations. Users that are not part of a group can't view the posts that are shared inside the group.
- **Search Singer:** User should be able to search for singers via search engine.
- **Search User:** User should be able to search for users via search engine.
- **Search Album:** User should be able to search for albums via search engine.
- **View Bio:** User should be able to view a short bio of each singer.
- **View Popular Songs:** User should be able to see the list of popular songs. This list can be filtered based on location (worldwide/nearby) and genre(all/a specific genre).
- **View Suggestions:** User should be able to see album and song suggestions based on genre.
- **View Statistics:** User should be able to see statistics about songs, albums, playlists and singers. Those statistics include information about number of listens and number of sells.


Visual Paradigm Standard(Bilkent Univ.)
Music Playing Database System
Sign-up
Accept Follow Request
View Album
Reject Follow Request
View Song
Create Group
Share Song
Share Playlist
Listen Music
Buy Album
Buy Item
Login
Like/Dislike Song
Create Playlist
Like/Dislike Playlist
Buy Song
Invite to Group
Make Comment
Search Singer
Follow User
Search User
Follow Playlist
View Statistics
Follow Singer
View Popular Songs
View Bio
View Suggestions
Search Album
<<Include>>
Music Fan

3.1.2) Music Producer

Music producers have the same authorization type with the music fan user type. Therefore they should be able to view the same content. This means they should be able to see statistics about songs, albums and singers. They should also be able to view information about popular songs based on location, date and genre. They can make use of these data in their production process and decisions.

- **Login:** User should login to system with a username and password.
- **Sign-up:** User should sign-up to system by registering his/her username and password.
- **View Album:** User should be able to see all albums of a singer in the database.
- **View Song:** User should be able to see all songs of a singer in the database. By navigatin in that menu, user can also see all songs in a particular album.
- **View Popular Songs:** User should be able to see the list of popular songs. This list can be filtered based on location (worldwide/nearby) and genre(all/a specific genre).
- **View Statistics:** User should be able to see statistics about songs, albums, playlists and singers. Those statistics include information about number of listens and number of sells.
- **Contact with Application Maintainers:** User should be able to contact with application maintainers.
- **Request Addition:** After contacting with the application maintainers, user should be able to request the addition of their songs and albums to application.



3.2 ALGORITHMS

Our system requires the registration of the user to the system, therefore the entered password will be checked in the program according to the following additional criteria to the SQL statement, which checks whether it is at least 6 character and the entered two passwords mathes or not:

- whether it contains one capital letter
- and whether it contains at least one special character.

These restrictions are used to make our system more secure.

In User page, Overview tab, there will be various tables that contain each user activity. We will write queries for each of these tables. Then, in the client side we will sort these different queries in chronological order. This is not possible in SQL as these tables cannot be unioned because they have different attributes. We will use a sorting algorithm. Same sorting algorithm will be used for main page for the user activities of the followed users, by using predefined views in the system. The most recent 10 results will be taken from each query and all of them will be put into a chronological order. Most recent 10 activities will be displayed on the main page. Same procedure will be used for the group overview tab which displays the user activities in the group.

Number of monthly listeners will be computed for each artist, on each click the counter will be incremented and by using the current system time in the program which was taken one month ago and one month later, the monthly listeners will be computed. Each month the counter will be reseted to 0.

For search engine on the main page, a searching algorithm will be used for each category, which are user, song, album and band. The results will be displayed for each category in different tabs.

Lastly, in order to provide a consistent and error free system, the logical errors should be prevented. These logical errors mostly occur due to the time attributes which are TIMESTAP, DATETIME; hence in our program we will make sure that each of them is valid during retrieval, deletion and modification of these attributes. For instance, we will use the time considering at most the minutes, therefore we will restrict our system.Additionally, in the follow_user relation we use since attribute which is a DATETIME, if this attribute is 0 then the system will know that user does not accept the follow request yet.

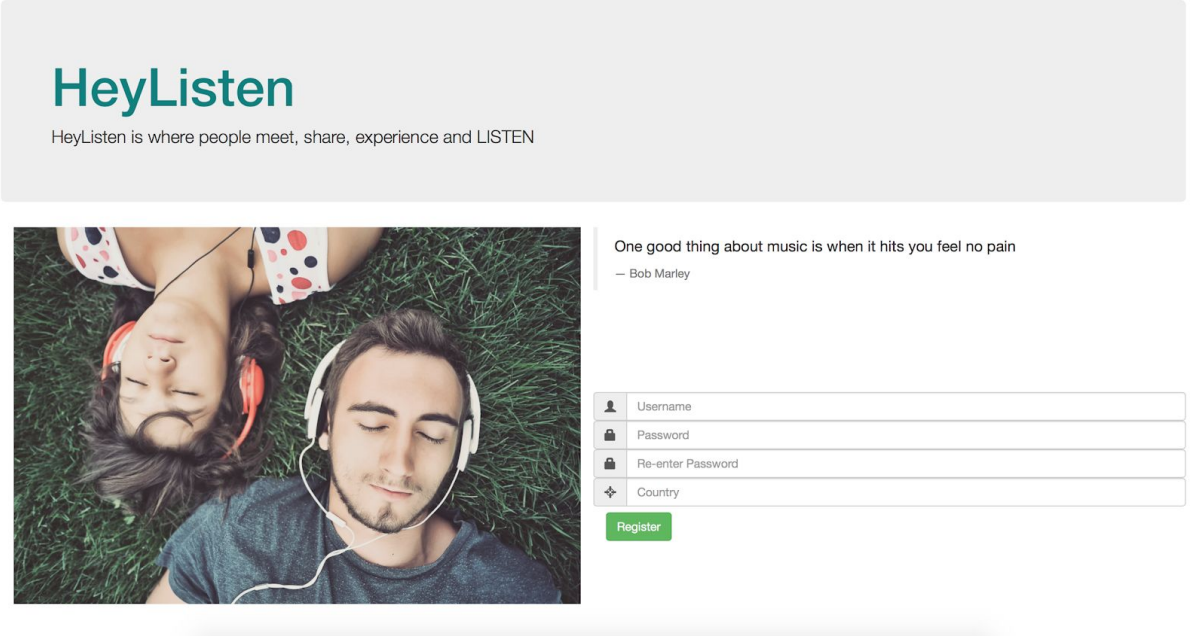
3.3 DATA STRUCTURES

As program sorts each recent user activities additional to the database tables, we will use a list to store the result of the queries.

In the relational model, numeric types are used for ID's of the entities and defined as INT. For username and name attributes variable length VARCHAR is used with maximum characters of 45. The time attributes are defined as TIMESTAMP while the date is defined as DATETIME. In the like relations type attribute and the isPrivate attribute of user and playlist is stored as a BIT(1). For type of the follow_like relation we used TINYINT as well as the price of the item.

4.0 USER INTERFACE DESIGN AND SQL STATEMENTS

4.1 REGISTER



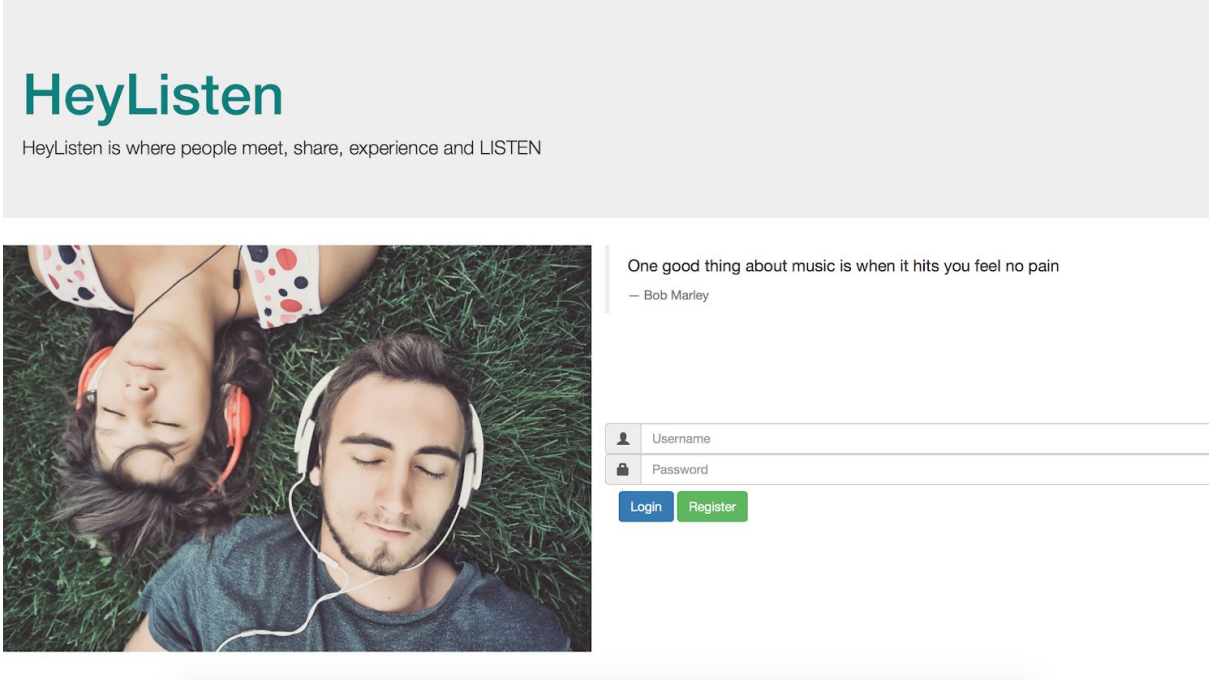
Inputs: @username @ password1 @password2 @country

Process: Users cannot use the system without registering if they do not have an account. Password will be asked twice for confirmation. It should be at least six characters.
Whether it has special characters or at least one capital character will be checked in the program.

SQL Statements:
// Insert the new user if password conditions are met

```
INSERT INTO user
VALUES (@username, @password1, 0, @country, 0)
WHERE @password1 = password2 and
            LIKE @password1 = '_ _ _ _ _ _ %' ;
```

4.2 LOGIN



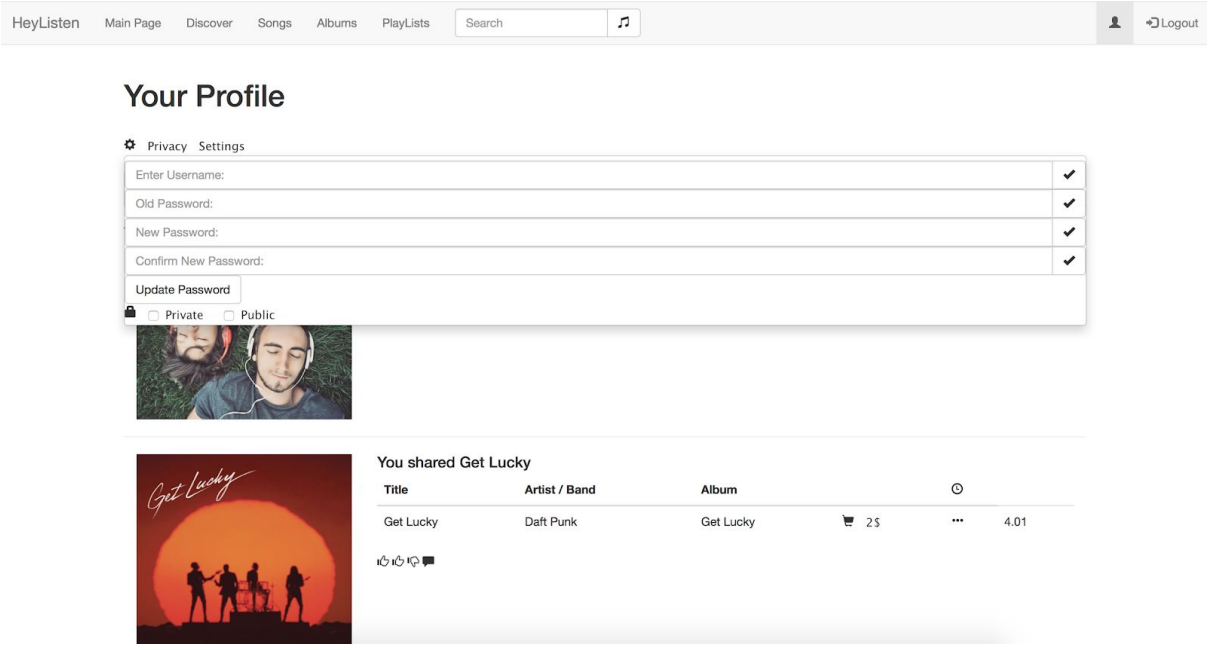
Inputs: @username @password

Process: Users need to login to system in order to see the functionality. Therefore they will provide their username and password to login. If user has not got an accout, can click on register button and go to register page.

SQL Statements:
// Check validity of credentials

```
SELECT username
FROM user
WHERE username = @username AND password = @password ;
```

4.3 CHANGE USER SETTINGS



Change Password:

Inputs: @username, @oldPassword, @newPassword1, @newPassword2
* @username is the username provided in the change settings form

Process: A pop up appears when user clicks a button in his/her profile to change password. There, user can change the password by providing the old password and the new password twice. Validity of the provided passwords are checked with SQL.

SQL Statements:
// Update user's password with the new password if conditions are met

UPDATE user
SET password = @newPassword
WHERE username = @ username AND password = @oldPassword AND
@newPassword1 = @newPassword2 AND
LIKE @newPassword1 = '_ _ _ _ _ _ %' ;

Change profile privacy:

Inputs: @username, @private

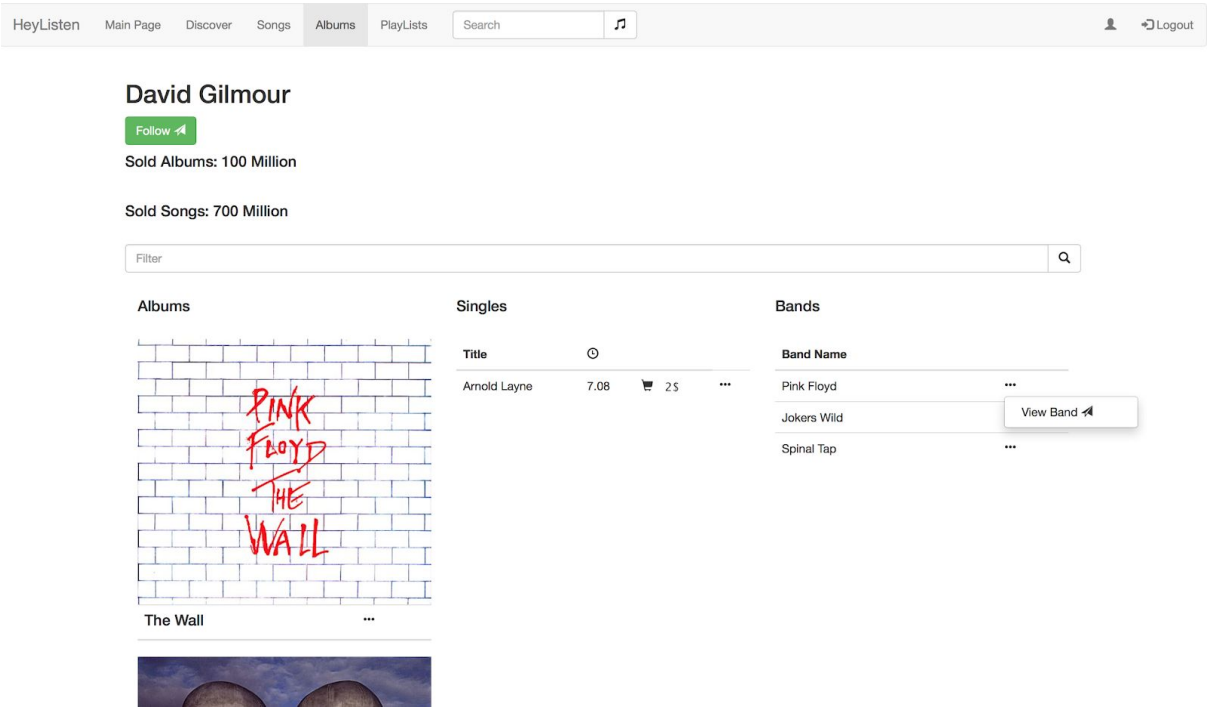
* @username is the current registered username

Process: A pop-up appears when user clicks a button in her profile to change the privacy of her profile. There, user can select private or not private and user table is updated accordingly.

SQL Statements:
// Update user table with new privacy setting

UPDATE user
SET isPrivate = @private
WHERE username = @ username ;

4.4 MUSICIAN PAGE



Inputs: @artistID

Process: When user clicks name of the musician on any page, she is redirected to musician's page. There, musician's name, bio, followers and social media connections are displayed. Moreover, Musician's albums, singles, bands, role (singer, guitarist etc.), number of total sold songs and albums are listed.

SQL statements:
//Display musician type, name, bio, connections

```
SELECT name, type, date_of_birth, place_of_birth
FROM musician
WHERE ID = @ artistID;


SELECT other_connections
FROM musician_connections
WHERE ID = @artistID;
```

// Display number of followers

```
SELECT COUNT(username)
FROM follow_artist
WHERE ID = @artistID ;
```

// When user clicks followers of the musician, display names of followers

```
SELECT username
FROM follow_artist
WHERE ID = @artistID ;
```

// Display albums

```
WITH tempAlbum(ID) AS
SELECT albumID
FROM has_Song,  song
WHERE artistID = @artistID AND songID = ID AND albumID is not null ;

SELECT ID,name
FROM album NATURAL JOIN tempAlbum ;
```

// Display singles

```
SELECT S.ID ,name
FROM has_Song,  song S , items I
WHERE artistID = @artistID AND songID = S.ID AND S.ID = I.ID AND albumID is
null ;
```

// Display band

```
SELECT name
FROM band_member, artist
WHERE musicianID = @artistID AND ID= bandID;
```

// Display number of sold songs

```
SELECT COUNT(*)
```

```
FROM buy
WHERE itemID IN (SELECT DISTINCT songID
                 FROM has_Song,  song
                 WHERE artistID = @artistID AND songID = ID ) ;
```

// Display number of sold albums

```
SELECT COUNT(*)
FROM buy
WHERE itemID IN
                (SELECT DISTINCT albumID
                 FROM has_Song,  song
WHERE artistID = @artistID AND songID = ID AND albumID is not null) ;
```

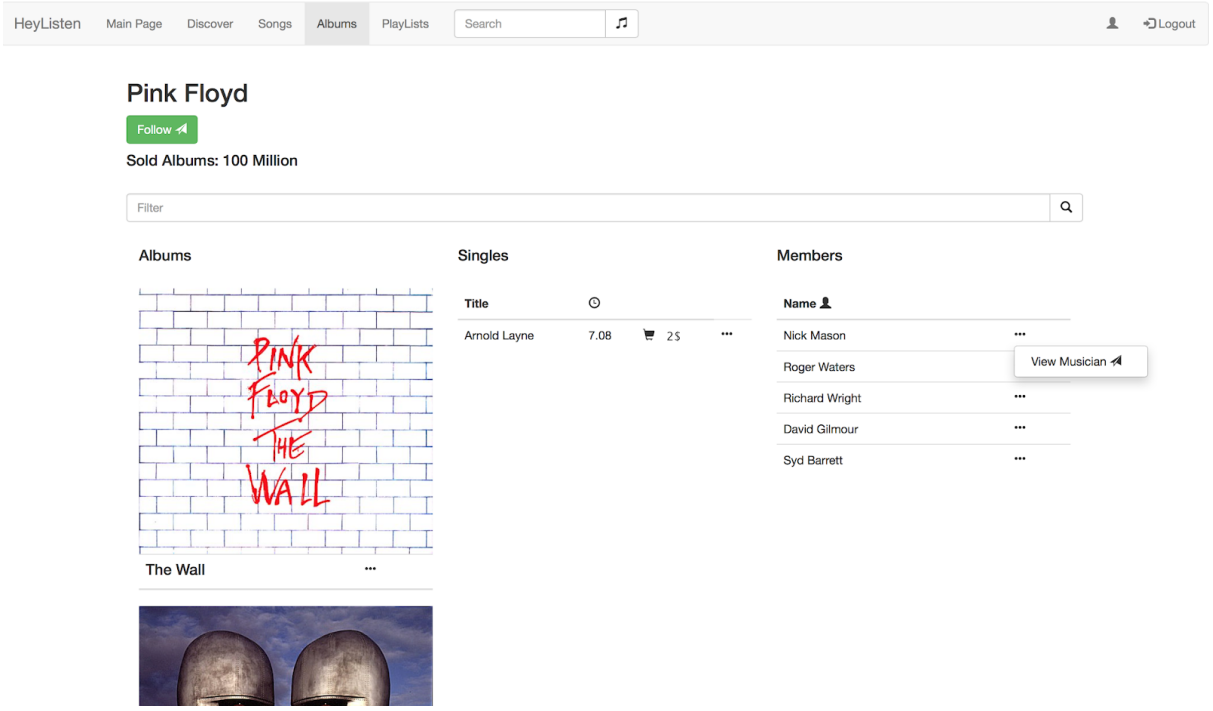
Process: When user clicks follow button on any musician page, user starts to follow the musician.

SQL statements:

```
INSERT INTO follow_artist

VALUES (@username, @artistID) ;
```

4.5 BAND PAGE



Input: @artistID

Process: When user clicks a band name on any page this page will come up. They can follow the band, see the musicians in the band and number of sold albums and songs, and the followers of the band. They can click to the albums/singles tab of band. They will see an information about the band and the establishment date of the band.

SQL statements:

// Display name of band, date of establishment and info

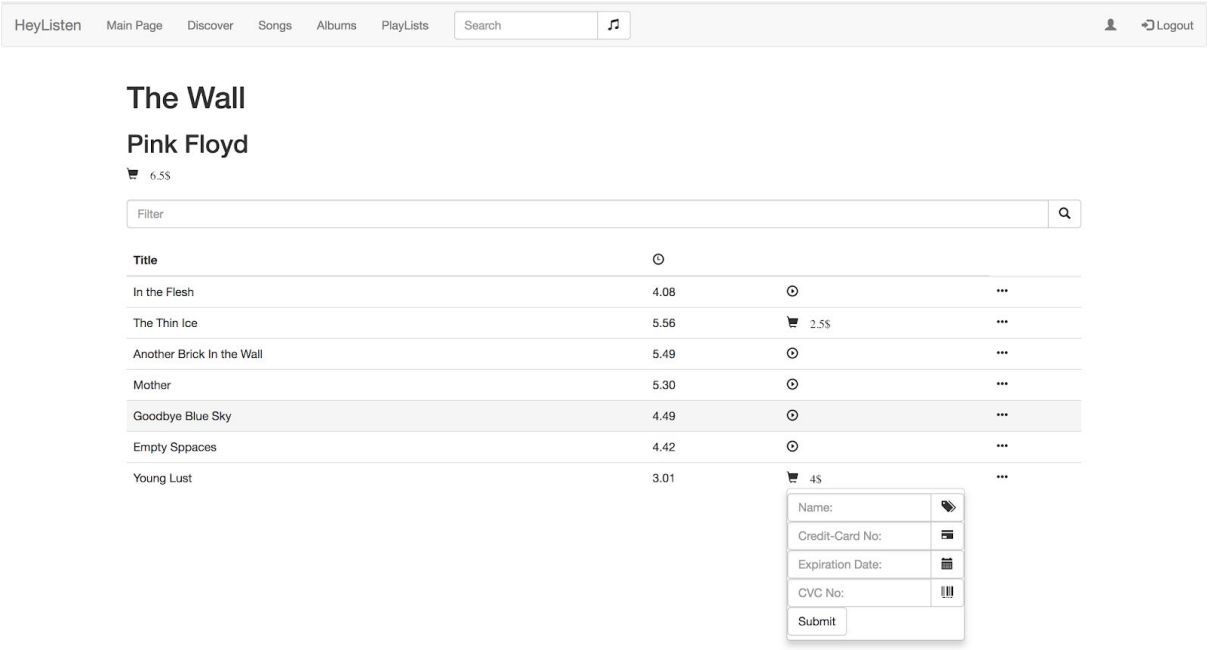
```
SELECT name, date_of_establishment, info_text
FROM artist NATURAL JOIN band
WHERE ID = @artistID;
```

// Display musicians in the band

```
SELECT ID, name
FROM band_member B, artist A
WHERE  B.bandID = @artistID AND B.musicianID = A.ID ;
```

Displaying albums, singles, number of sold song and sold albums have the same SQL statements with musician page.
When user clicks follow button on band page same SQL statement applies

4.6 ALBUM PAGE



Inputs: @username **,**@itemID, @time, @bought_for , @name, @songID
*@username is current username

Process: When user clicks an album name on any page, this page will be displayed. List of songs, which belongs to the album will be displayed. User can buy these songs and afterwards can add them to their playlists. The name and price of the song will be displayed.

SQL statements:

// Display list of songs

```
SELECT I.name, I.price, I.ID
FROM song S, items I
WHERE S.ID = I.ID AND S.albumID = @albumID ;
```

// Buy song or album

```
INSERT INTO buy
VALUES (@username, @item_ID, @bought_for, @time) ;
```

// Add song to playlist
// A pop up will occur and allow user to select which playlist to add.

```
INSERT INTO playlist_includes
VALUES (@username, @name, @songID)
WHERE (@username,@songID ) IN ( SELECT username, itemID
                                FROM buy
    WHERE bought_for = @username);
```

Process: A pop up will appear after clicking buy button. User will enter name of the owner, their card number, its expiration date and CVC number. Afterwards,user will select whether s/he buys himself or to the one they follow.

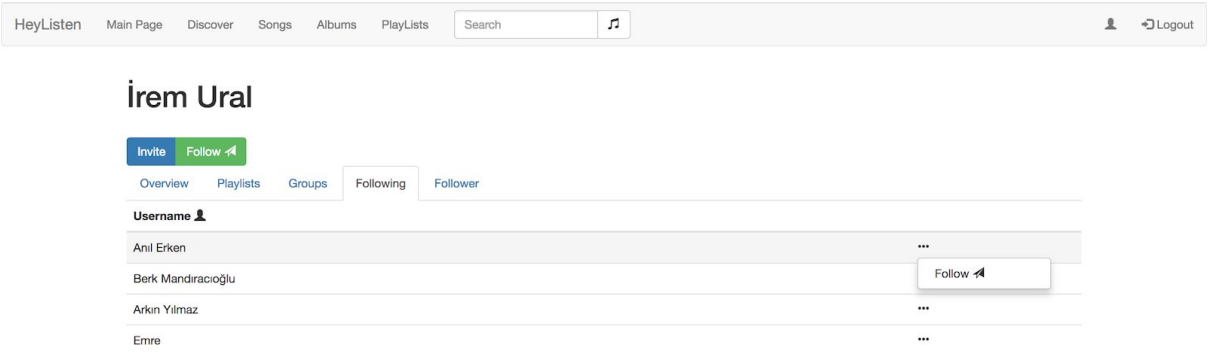
4.7 USER PAGE

Inputs: @username, @profilename

Process: When user clicks on profile, this page will be displayed. In Overview tab, user can view her own activities in a chronological order. For example, user can view songs, playlists that are shared by her and view her comments and likes on posts. In Playlists tab, user can view playlists that are created by her. In Groups tab, user can view groups that are created by her or groups that she is a member. In Followers

tab, user can view the users that are following her. In Following tab, user can view the users that she is following.

SQL Statements (with corresponding user interfaces):

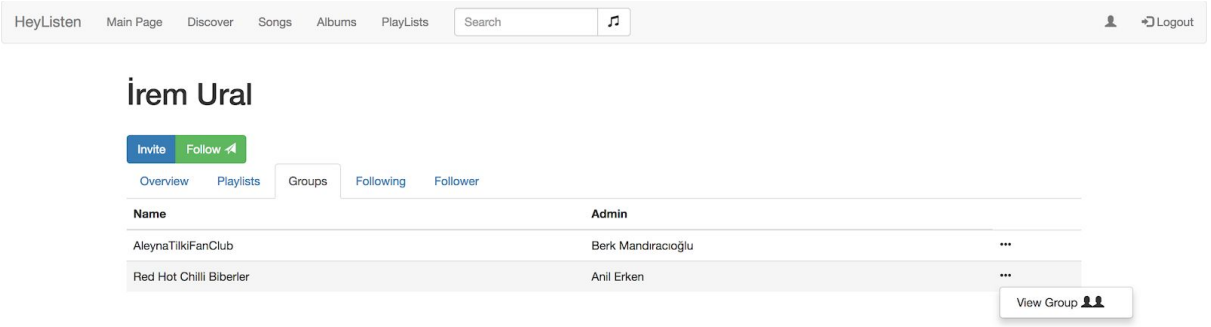


// Followers Tab displays followers

```
SELECT follower_id
FROM follow_user
WHERE followed_id = @username ;
```

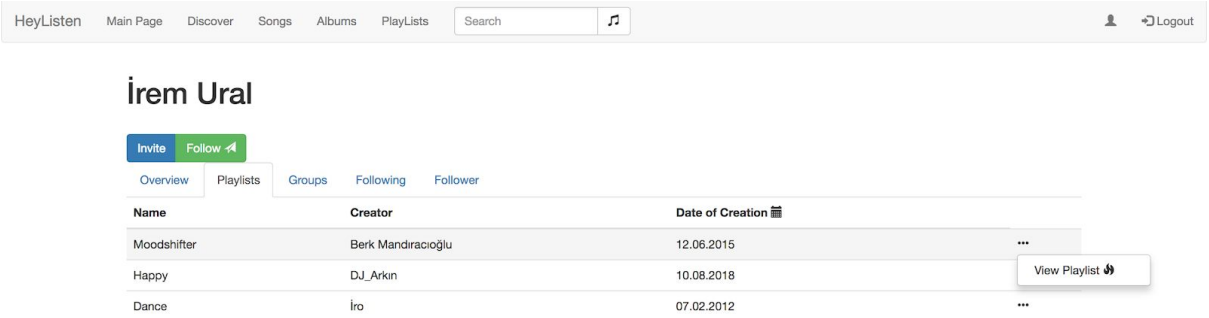
// Following Tab displays followed users

```
SELECT followed_id
FROM follow_user
WHERE follower_id = @username;
```



// Groups Tab displays the groups that user joined

```
(SELECT g1.ID, g1.name
FROM group g1
WHERE g1.admin = @username)
UNION
(SELECT g2.ID, g2.name
FROM invite_user m, group g2
WHERE m.receiver = @username
AND m.groupID = g2.ID
AND m.decision = 1)
```



// Playlists Tab displays playlists of the user

```
SELECT name
FROM playlist
WHERE username = @username
```

// Display number of followers

```
SELECT COUNT(*)
FROM follow_user
WHERE followed_id = @username  AND since <> 0;
```

// Display number of followings

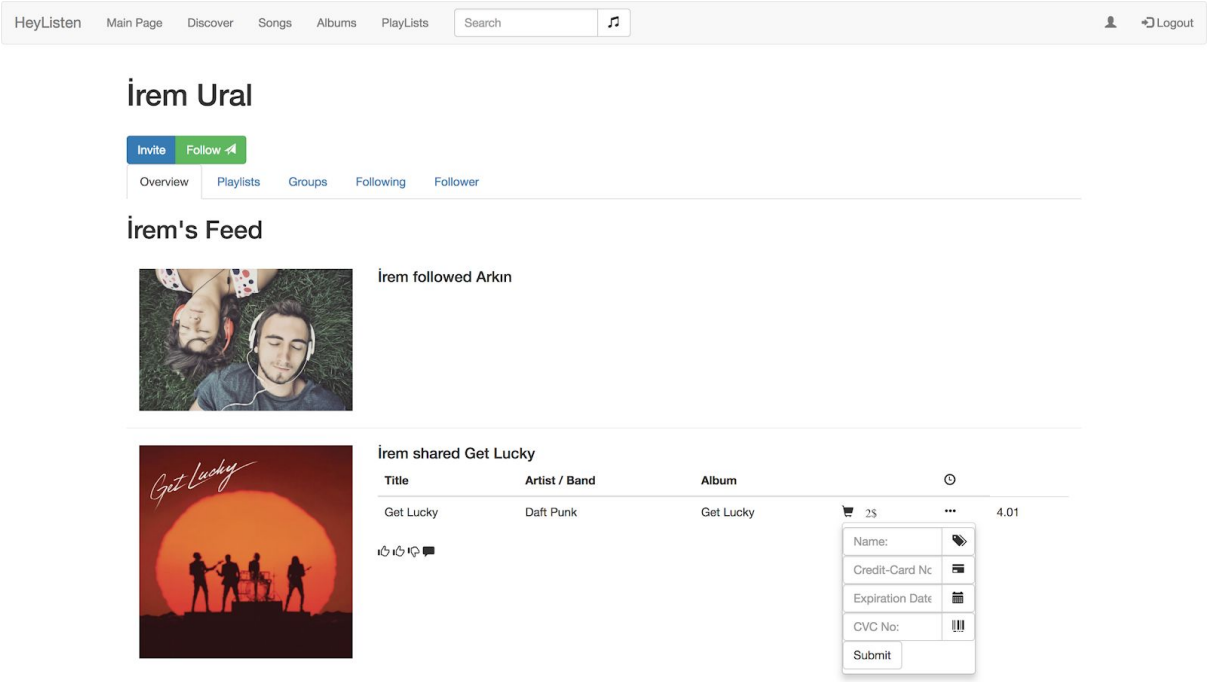
```
SELECT COUNT(*)
FROM follow_user
WHERE follower_id = @username AND since <> 0;
```

Process: User can click a button is his/her own profile to create a new group. He/she will be the admin of the group and can invite people on it.

Inputs: @username, @name

// Create new group

```
INSERT INTO group
VALUES (DEFAULt, @username,@name ) ;
```



// Display playlists that user shared in her profile

```
CREATE VIEW playlistPostOnUserPage( name, username, time, share_time)  AS
        SELECT P2.name, P2.username, P2.time, P3.share_time
        FROM playlist P2, post_playlist P1, post P3
        WHERE ( P1.p_name, P1.p_username ) = (P2.name, P2.username) AND
        P3.ID =  P1,postID AND P3.username = @username
        ORDER BY P3.share_time DESC
        LIMIT 10 ;
```

// Display songs that user shared in her profile

```
CREATE VIEW songPostOnUserPage( name, duration, type, share_time)  AS
        SELECT I.name, S.duration, S.type , P2.share_time
        FROM post_song P1, items I, song S, post P2
        WHERE I.ID = P1.s_ID  AND  I.ID = s.ID  AND P2.ID =  P1.p_ID AND
                        P2.username = @username
        ORDER BY P2.share_time DESC
        LIMIT 10 ;
```

// Display playlists that user liked/disliked or followed

```
CREATE VIEW userFollowPlaylist(userUsername, creatorName, time, type) AS
SELECT userUsername, creatorName, time, type
FROM follow_like_playlist
WHERE userUsername = @username
ORDER BY time DESC
LIMIT 10 ;
```

// Display songs and albums that user liked or disliked

```
CREATE VIEW userlikesItem(username, name, time, type) AS
SELECT itemID, I.name ,time, type
FROM like_item, items I
WHERE I.ID = itemID AND username = @username
ORDER BY time DESC
LIMIT 10 ;
```

// Display that user followed an artist

```
CREATE VIEW userFollowsArtist(username, ID ,nameOfArtist) AS
SELECT username, ID, A2.name
FROM follow_artist A, artist A2
WHERE A.ID = A2.ID AND username = @username
ORDER BY A.time DESC
LIMIT 10 ;
```

// Display that user followed another user

```
SELECT F.followed_id
FROM follow_user F
WHERE F.since <> 0 AND F.follower_id = @username
ORDER BY F.since DESC
LIMIT 10;
```

PROCESS: User can follow a nonprivate user without any acceptence. However, in order to follow a private user, the request should be accepted by the other user. Until s/he accepts the request since attribute is set to 0.

INPUTS: @profileName, @ since, @username

SQL Statements:

```
SELECT isPrivate
FROM user
WHERE username = @profileName
```

If isPrivate is 0 then we will use the following:

```
INSERT INTO follow_user
        VALUES (@username, @profileName, @since)
```

else this statement will be used

```
INSERT INTO follow_user
VALUES (@username, @profileName, 0) ;
```

-This decision will be made through the JavaScript code.

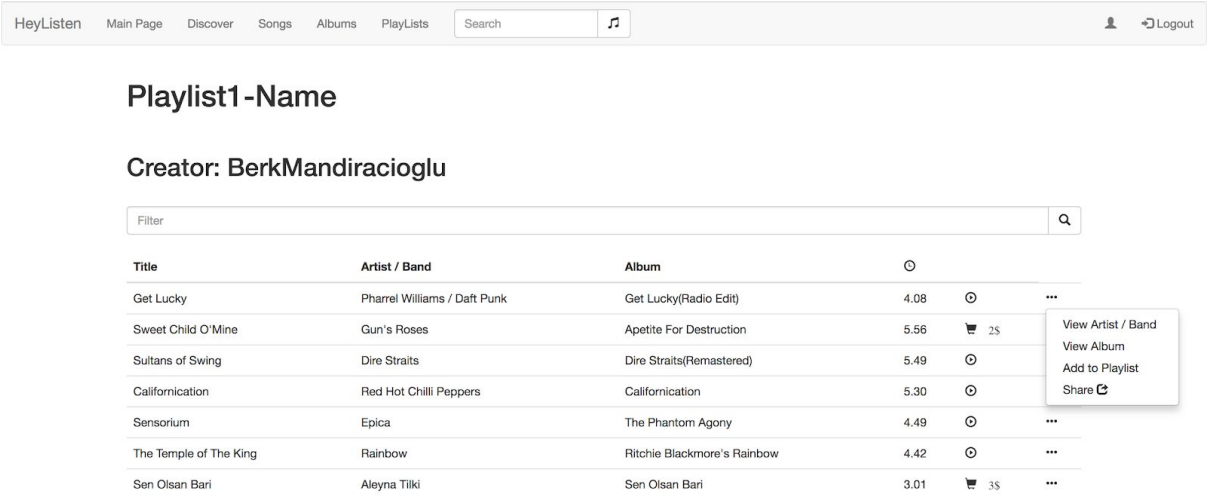
Input: @since will be the current date taken from system.

```
//accept follow request
UPDATE follow_user
SET since = @since
 WHERE   followed_id = @ username AND follower_id = @profileName;
```

// if the user rejects the follow request, the tuple will be deleted

```
DELETE FROM follow_user
WHERE follower_id = @followerName AND followed_id = @profileName;
```

4.8 PLAYLIST PAGE



Inputs: @username, @name

Process: When user open a playlist page. Songs in the playlist will be displayed, with its singer, duration.

SQL statements:

```
SELECT I.name,
      Ar.name,
      ( CASE
         WHEN S.albumID IS NULL THEN "no album"
```

```
        ELSE (SELECT name
              FROM   items
              WHERE  S.albumID = ID)
      END ),
     S.duration
FROM   playlist_includes P, song S, items I, artist Ar, has_song H
WHERE  P.songID = S.ID AND I.ID = S.ID
      AND Ar.ID = H.artistID AND H.songID = S.ID
GROUP  BY I.name, Ar.name, S.albumID, S.duration
HAVING Ar.ID IN(SELECT bandID
             FROM   band_member)
      OR Ar.ID NOT IN(SELECT musicianID
                   FROM   band_member)
```

// Make playlist private

```
UPDATE playlist
SET isPrivate= @isPrivate
WHERE username = @ username AND name = @name ;
```

// Create new playlist

Inputs: @username, @time,@name

```
INSERT INTO playlist values(@username, @name, @time, 0)
```

4.9 SHARE, LIKE , COMMENT(ACTIVITIES)

Process: Users can share playlists, songs and albums. In order to share a song user selects the song they want to share from her own song library or a playlist or an album. Users can share playlists from a User page where playlists are listed or from a Group page where playlists are listed. These shared posts can appear on a Group page if they were shared in a group, otherwise these posts can be viewed from main page if the user follows the poster. Moreover, users can select an album they want to share from the Album page. User can like, dislike or make comment on shared posts or singular items(album, song) or playlists. The activities of a person can be viewed in Main page of the user if the user follows the person.

Inputs: @time and @username

// Share a post

```
INSERT INTO post VALUES(DEFAULT, @time, @username);
```

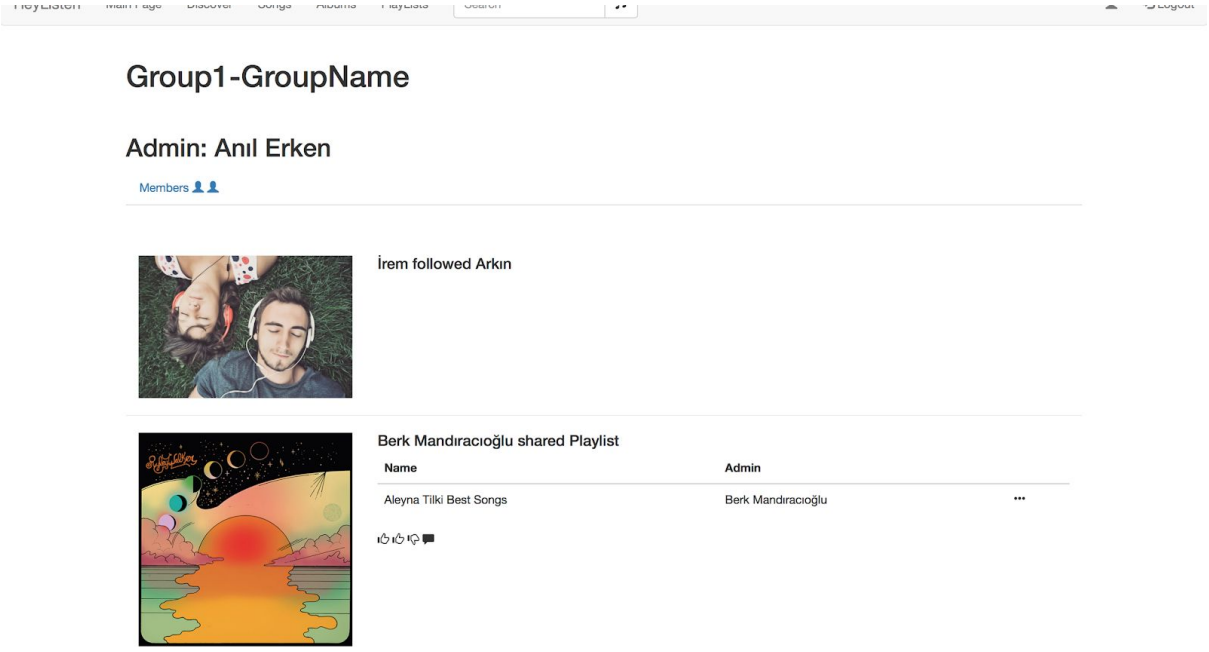
// Make comment
Inputs: @time, @text
INSERT INTO comment VALUES(DEFAULT, @time, @text);

// Like or dislike a post
Inputs: @username,@time, @type, @postID
INSERT INTO like_post VALUES (@username, @postID, @time. @type);

// Like or dislike comment
Inputs: @username,@commentID, @type, @time
INSERT INTO like_comment VALUES (@username, @commentID, @time. @type);

// Like or dislike a song or album
Inputs: @username,@itemID, @type, @time
INSERT INTO like_item VALUES (@username, @itemID, @time. @type);

4.10 GROUP PAGE



Inputs: @groupID

Process: Name of the group and the admin will be displayed in this page. Playlist of the group is displayed and the members of the group will be displayed in another tab. Recent posts of the group will be displayed in the overview tab.

SQL statements:

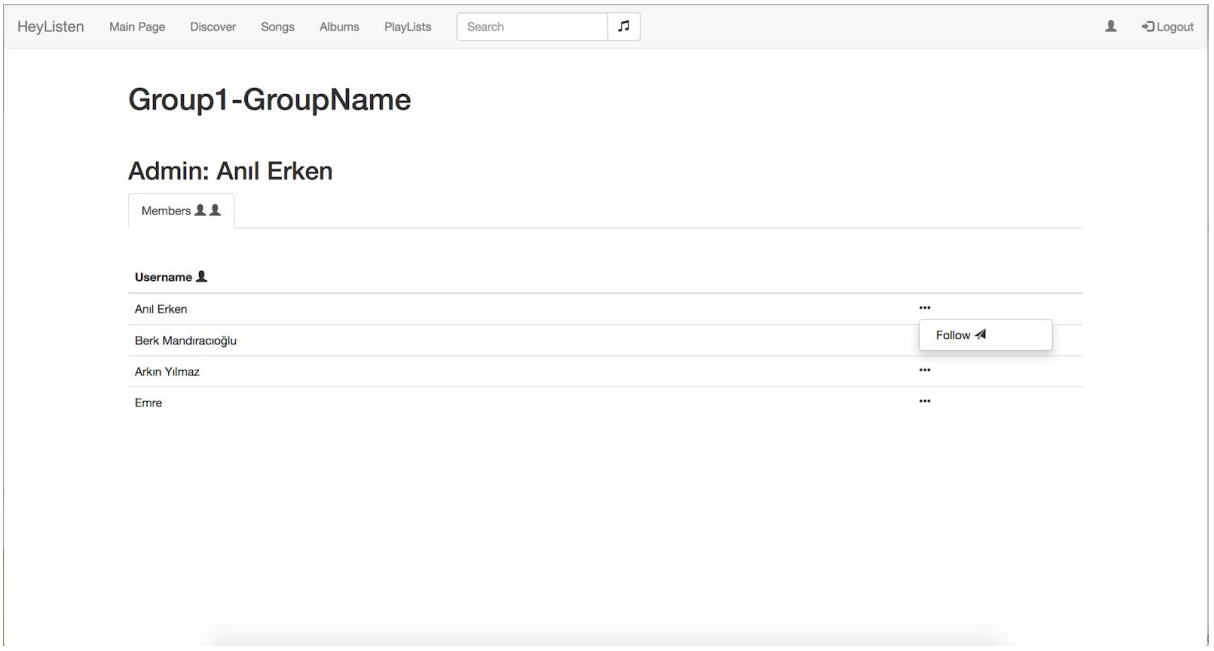
// Display name of the group and admin

```
SELECT admin, name
FROM     group
WHERE  groupID = @groupID
```

// Display playlist of the group

```
SELECT P.username, P.name
FROM playlist P
WHERE (@username, P.name) = (SELECT DISTINCT admin, name
                             FROM     group
                             WHERE  groupID = @groupID ) ;
```

This will be used to direct user to playlist page, the songs are displayed in the same way in the playlist page.



// Display members of the group

```
SELECT reciever
FROM invite_user
WHERE groupID = @groupID AND decision = 1
```

// Display number of members

```
SELECT COUNT(*)+ 1
FROM invite_user
WHERE groupID = @groupID AND decision = 1
```

// Recent posts in group:

```
WITH postPlaylist (postID, p_name, p_username) AS
        SELECT post_ID, p_name, p_username
        FROM  post_playlist
        WHERE postID  = ( SELECT postID
                          FROM share_in_group
                          WHERE groupID = @groupID);

WITH postSong(p_D, s_ID) AS
        SELECT p_ID, s_ID
        FROM    post_song
        WHERE p_ID = ( SELECT postID
                       FROM share_in_group
                       WHERE groupID = @groupID);
```

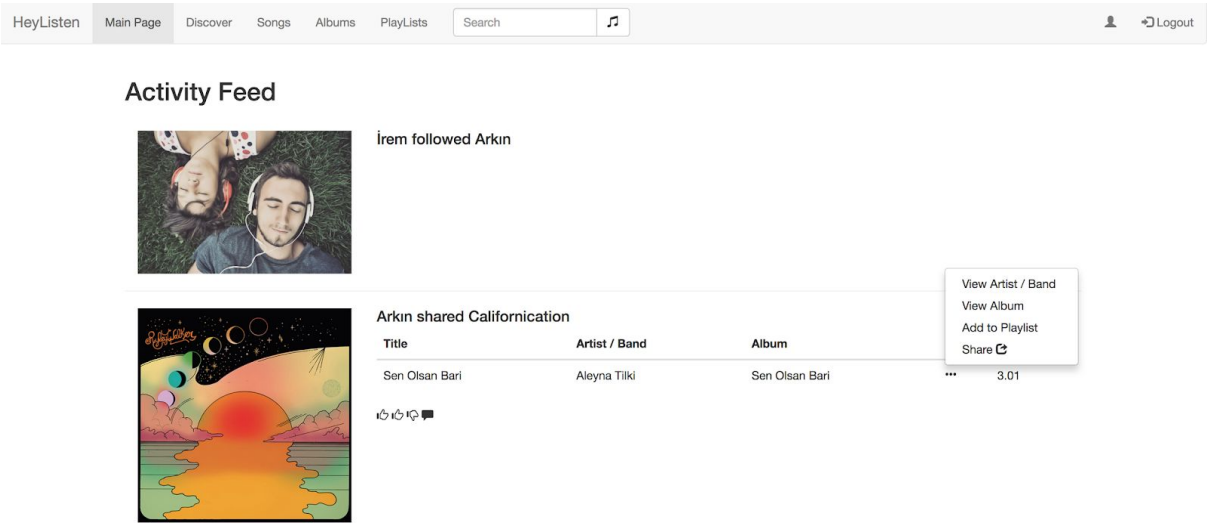
// Shared playlists in the group

```
CREATE VIEW playlistPost(name, username, time)  AS
        SELECT P2.name, P2.username, P2.time
        FROM postPlaylist P1, playlist P2
        WHERE ( P1.p_name, P1.p_username ) = (P2.name, P2.username);
```

// Shared songs in the group

```
CREATE VIEW songPost(name, duration, type)  AS
        SELECT I.name, S.duration, S.type
        FROM postSong P1, items I, song S
        WHERE I.ID = P1.s_ID  AND  I.ID = s.ID ;
```

4.11 MAIN PAGE



Input: @username

Process: The main page appears when user log on to system. Here, friend activities are displayed. These include shared playlists, shared songs and follow/like activities on playlists, artist, songs, other users and albums.

SQL Statements:

// A view is used to get usernames that current user followed

```
CREATE VIEW followedUsers(followed_ID) AS
SELECT followed_id, since
FROM follow_user
WHERE follower_id = @username AND since <> 0
ORDER BY since DESC
```

// Playlists that shared by a user that current user follows

```
CREATE VIEW playlistPostOnMainPage( name, username, time, share_time)  AS
        SELECT P2.name, P2.username, P2.time, P3.share_time
        FROM playlist P2, post_playlist P1, post P3
        WHERE ( P1.p_name, P1.p_username ) = (P2.name, P2.username) AND
        P3.ID =  P1,postID AND P3.username IN ( SELECT *
    FROM followedUsers)
        ORDER BY P3.share_time DESC
```

```
        LIMIT 10 ;
```

// Songs that shared by a user that current user follows

```
CREATE VIEW songPostOnMainPage( name, duration, type, share_time)  AS
        SELECT I.name, S.duration, S.type , P2.share_time
        FROM post_song P1, items I, song S, post P2
        WHERE I.ID = P1.s_ID  AND  I.ID = s.ID  AND P2.ID =  P1.p_ID AND
                        P2.username IN ( SELECT *  FROM followedUsers)
        ORDER BY P2.share_time DESC
        LIMIT 10 ;
```

// Display that a user that current user follows followed/liked/disliked a playlist

```
CREATE VIEW followPlaylist(userUsername, creatorName, time, type) AS
SELECT userUsername, creatorName, time, type
FROM follow_like_playlist
WHERE userUsername IN ( SELECT *  FROM followedUsers)
ORDER BY time DESC
LIMIT 10 ;
```

// Display that a user that current user follows liked/disliked a song or album

```
CREATE VIEW followItem(username, name, time, type) AS
SELECT itemID, I.name ,time, type
FROM like_item, items I
WHERE I.ID = itemID AND username IN ( SELECT *  FROM followedUsers)
ORDER BY time DESC
LIMIT 10 ;
```

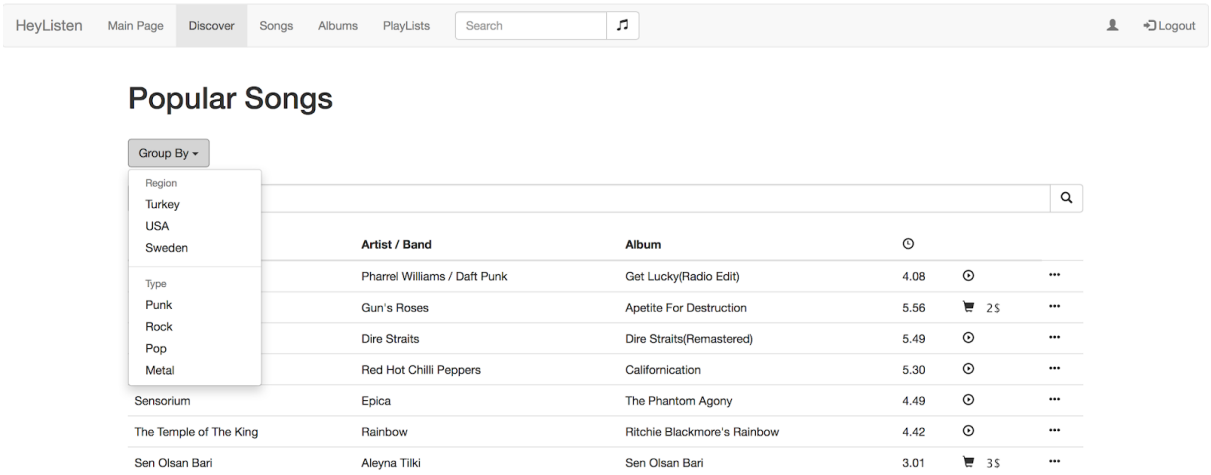
// Display that a user that current user follows followed an artist

```
CREATE VIEW followArtist(username, ID ,nameOfArtist) AS
SELECT username, ID, A2.name
FROM follow_artist A, artist A2
WHERE A.ID = A2.ID AND username IN ( SELECT *  FROM followedUsers)
ORDER BY A.time DESC
LIMIT 10 ;
```

// Display that a user that current user follows, followed a user

```
SELECT F.follower_id, F.followed_id
FROM follow_user F
where F.follower_id in ( SELECT * FROM followedUsers ) AND F.since <> 0
ORDER BY F.since DESC
LIMIT 10;
```

4.12 DISCOVER PAGE



Input: @region

Process: In discover page, list of popular songs are displayed according to region and type. User can select type or region. In each category top 100 songs are listed.

SQL Statements:

// Show regional

```
SELECT COUNT(*) AS COUNT, itemID, I.name
FROM buy, song S , items I
WHERE itemID not in (SELECT ID
   FROM album  ) AND  itemID = s.ID AND s.ID = I.ID AND
   region = @region
GROUP BY itemID
ORDER BY COUNT DESC
LIMIT 100;
```

// Show worldwide

```
SELECT COUNT(*) AS COUNT, itemID, I.name
FROM buy, song S , items I
WHERE itemID not in (SELECT ID
```

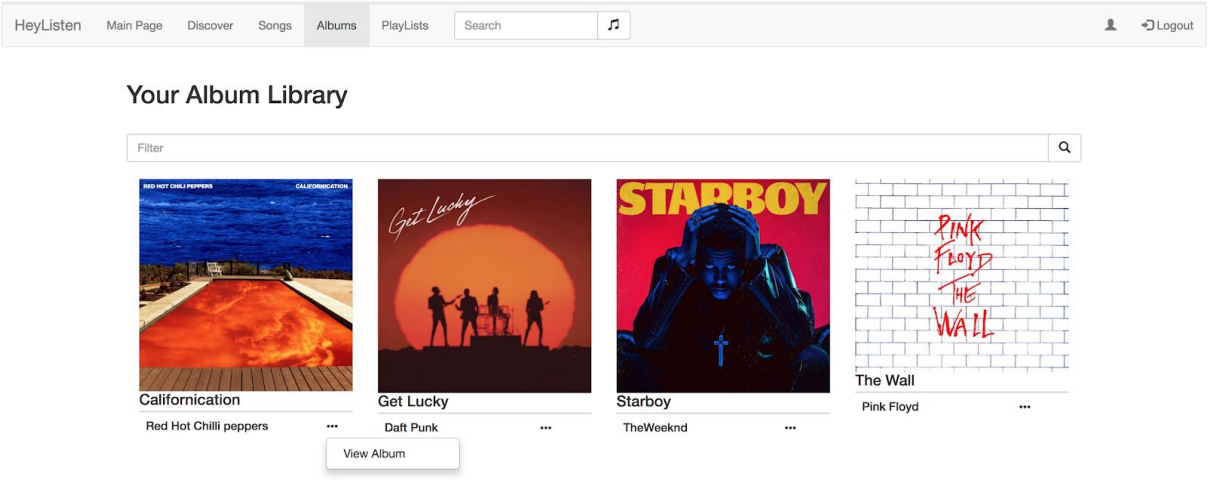
```
   FROM album  ) AND  itemID = s.ID AND s.ID = I.ID
GROUP BY itemID
ORDER BY COUNT DESC
LIMIT 100;
```

//Show according to type of the song

Input: @type

```
SELECT COUNT(*) AS COUNT, itemID, I.name
FROM buy, song S , items I
WHERE itemID not in (SELECT ID
   FROM album  ) AND  itemID = s.ID AND s.ID = I.ID AND
   type = @type
GROUP BY itemID
ORDER BY COUNT DESC
LIMIT 100
```

4.13 ALBUMS PAGE



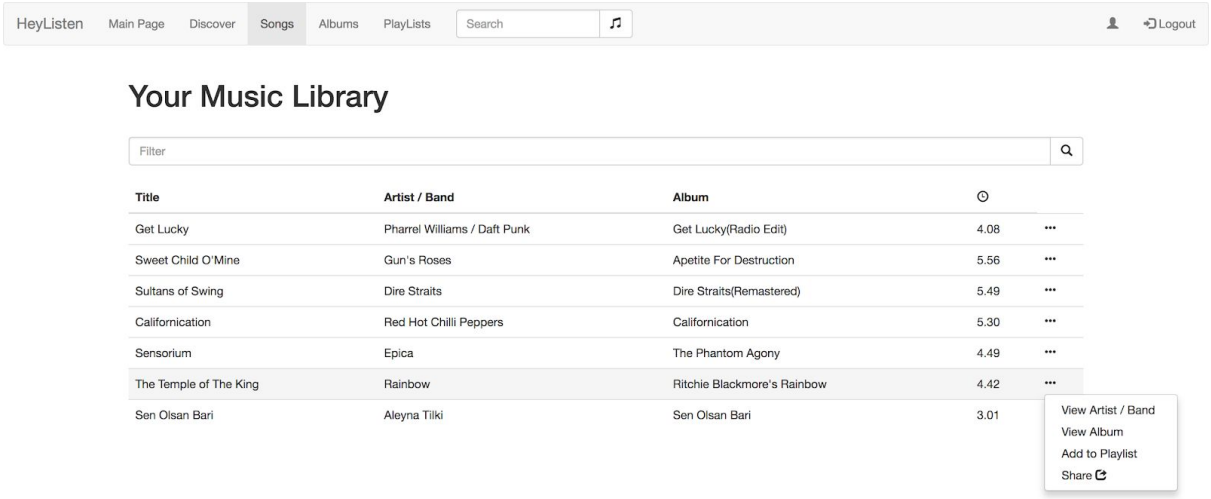
Input: @username

Process: In Albums Page, user can view the albums that she bought for herself or albums that are bought for her. This means that she can listen these albums and can view them listed in Albums tab.

SQL Statement:

```
SELECT A.ID, I.name, Ar.name, A.coverpath
FROM items I, Artist Ar, has_Song H, Album A ,Song S, buy B
WHERE I.ID = A.ID and Ar.ID = H.artistID
        and H.songID = S.ID and S.albumID = A.ID
        and I.ID = B.itemID and bought_for = @username
Group By I.name, Ar.name, A.ID, A.coverpath
HAVING Ar.ID in( select bandID from band_member)
        or Ar.ID not in(select musicianID from band_member)
```

4.14 SONGS PAGE



Inputs: @username

Procedure: User can view the songs that she bought for her or other bought for her. These songs are listed in the Songs Page. User can select these songs to share, add to playlist, view the artist and view the album.

SQL Statements:

```
SELECT I.name,
```

```
      Ar.name,
      ( CASE
         WHEN S.albumID IS NULL THEN "no album"
         ELSE (SELECT name
               FROM   items
               WHERE  S.albumID = ID)
       END ),
      S.duration
FROM   buy B, song S, items I, artist Ar, has_song H
WHERE  B.itemID = S.ID AND I.ID = S.ID
       AND Ar.ID = H.artistID AND H.songID = S.ID
       AND B.bought_for = @username
Group By I.name, Ar.name, S.albumID, S.duration
HAVING Ar.ID IN(SELECT bandID
              FROM   band_member)
       OR Ar.ID NOT IN(SELECT musicianID
                    FROM   band_member)
```

5. ADVANCED DATABASE COMPONENTS

5.1 REPORTS

Music production company can see some statistics about the artist, therefore from time to time we need recompution of these statistics. Hence the following reports are defined:

Input: @artistID

1. Display number of followers of an artist

```
SELECT COUNT(username)
         FROM follow_artist
         WHERE ID = @artistID ;
```

2. Display number of sold songs

```
SELECT COUNT(*)
FROM buy
WHERE itemID IN (SELECT DISTINCT songID
                 FROM has_Song,  song
                 WHERE artistID = @artistID AND songID = ID ) ;
```

3. Display number of sold albums

```
SELECT COUNT(*)
FROM buy
WHERE itemID IN
                (SELECT DISTINCT albumID
                 FROM has_Song,  song
```

```
WHERE artistID = @artistID AND songID = ID AND albumID is not null) ;
```

4. Display number of listeners of each song

```
SELECT listenCount
FROM    artist A, has_Song S, song S2
WHERE A.ID = @artistID AND S.artistID = A.ID AND S.songID = S2.ID;
```

5.2 VIEWS

View are stored in the system by its definition and when it is used the contents are dynamically created, recomputed again. As user activities are changing all the time we used views for the main page while displaying the most recent user activities of the followed users.The queries which are used in this process can be seen in the 4.11 MAIN PAGE section in the SQL statements part.

The shared playlists, songs, the follow actions displayed by the user, likes dislikes of songs/albums or playlists are computes in these views.

Likewise, for the user same activities are computed in different views, which are available in 4.7 USER PAGE section in SQL statements part.

Lastly, for group overview tab which has most recent posts in the group, shared playlists and songs in the group are displayed. the views created for each is written in 4.10 GROUP PAGE section in SQL statements part.

5.3 TRIGGERS

1- When user buys an item and the buy entity is updated, the budget of the user should be decremented in the same amount.

2- When user makes a comment a trigger is activated and it adds commentID to comment_on_post, comment_on_playlist or comment_on_item relation.

3- When a user deletes a comment trigger is activated and it deletes respected row of comment in comment_on_post,comment_on_playlist or comment_on_item relation relation.

4-When a user shares a post a trigger is activated and that post is also inserted into post_item or post_playlist depending on post type.

5- When user dislikes a post which was liked before or vice versa, a trigger will be used and it will update the existing tuple. This will be applied for comments and items as well.

5.4 NAMED PROCEDURES

We believe that reusing and recycling code is critical point and we care for number of lines of code that run in every page. Therefore, in HeyListen we will use procedures for queries instead of writing a new line of code every time we need to write a query. These procedures will be very similar to methods and functions. They will take parameters that a query needs and execute it using the given parameters. For example, we will write a procedure for INSERT INTO query which will take the table parameter and variable number of parameters for the values to be inserted. This way we will utilize code reuse and minimize number of lines in every page. Moreover, since our pages will be dynamic and we are collecting the data from the databases, we will write procedures to loop over the rows of the resulting query. While the procedure iterates over the rows it will write the necessary html elements to view these rows of data dynamically. Therefore, we will again utilize code reuse by writing a procedure for iterating over multi-rowed query results. We have many interfaces that list data as tables in HeyListen so procedure like above will be very useful.

5.5 CONSTRAINTS

1. Users must register to the system to use this web based application. If they already have an account they should first login to the system.
2. User must buy the song in order to listen it.
3. There can be at most 10 posts in the overview pages.
4. Posts can have either song or playlist.
5. One post can be shared in group or in the main page.
6. User cannot post empty posts or comments.
7. User cannot see the activities of the other user if the profile is kept private.
8. Time attributes of the entities will consider at most the minutes.
9. For group invitation decision can be 0,1 or 2. 0 for denial, 1 for acceptance, 2 for not responding.
10. For playlist follow_like type attribute can be 0, up to 5.

- 0 for follow
- 1 for follow and like
- 2 for follow and dislike
- 3 for like an unfollowed playlist
- 4 for dislike an unfollowed playlist

For the above in the cases 3 and 4, playlist must not be private.

6.0 IMPLEMENTATION PLAN

We are planning to use PHP, Javascript/Angular/Jquery. Bootstrap will be used for the view. HTML and CSS will be used for front-end development. We plan to use Javascript, PHP or JAVA for the server side. We will use MySQL for the database management.

7.0 WEBSITE

https://berkmandiracioglu.github.io/HeyListen/