Music Playing Database System

Project Design Report

Group 12

Alba Mustafaj 21500009

Argert Boja 21503525

Rubin Daija 21500010

Ndriçim Rrapi 21500342

Table of Contents

1.REVISED E/R MODEL

The feedback was very significant in improving the E/R model of our database and enhancing it for more complexity. The previous model was modified in the following way:

- temp_users was removed as it was unnecessary and redundant. Instead, a user entity with disjoint specialisation was created. Concretely, there three types of users: artist, production manager and consumer.
- Instead of having a "shares on wall relation", separate entities were constructed for comments and posts. They are week entities relying on user entity and having the corresponding likes and post on relations.
- A ternary relation named creates was put among production manager, artist and album. It demonstrates that artists and managers can create many albums but one album can be created by only one artists (which can add other contributors) but it can have different production managers.
- The unnecessary ternary purchase relation between artist, song and consumer was removed. Instead a binary purchase relation was put between consumer and song. Total participation constraint was fixed between album and song as well.
- The unnecessary "owns" relationship between consumer and song was removed as it was redundant. A new relation named report (which has a reason) was put between them to demonstrate that a consumer can report a song if he considers it inappropriate.
- The cardinality in the "forms" relation (previously called constructs) between consumer and playlist was corrected to 1:M and a "follows_playlist" and "rate" relation was added between them. Total participation constraint to demonstrate that a playlist cannot be empty was put in playlist_contains (previously named has) relation.

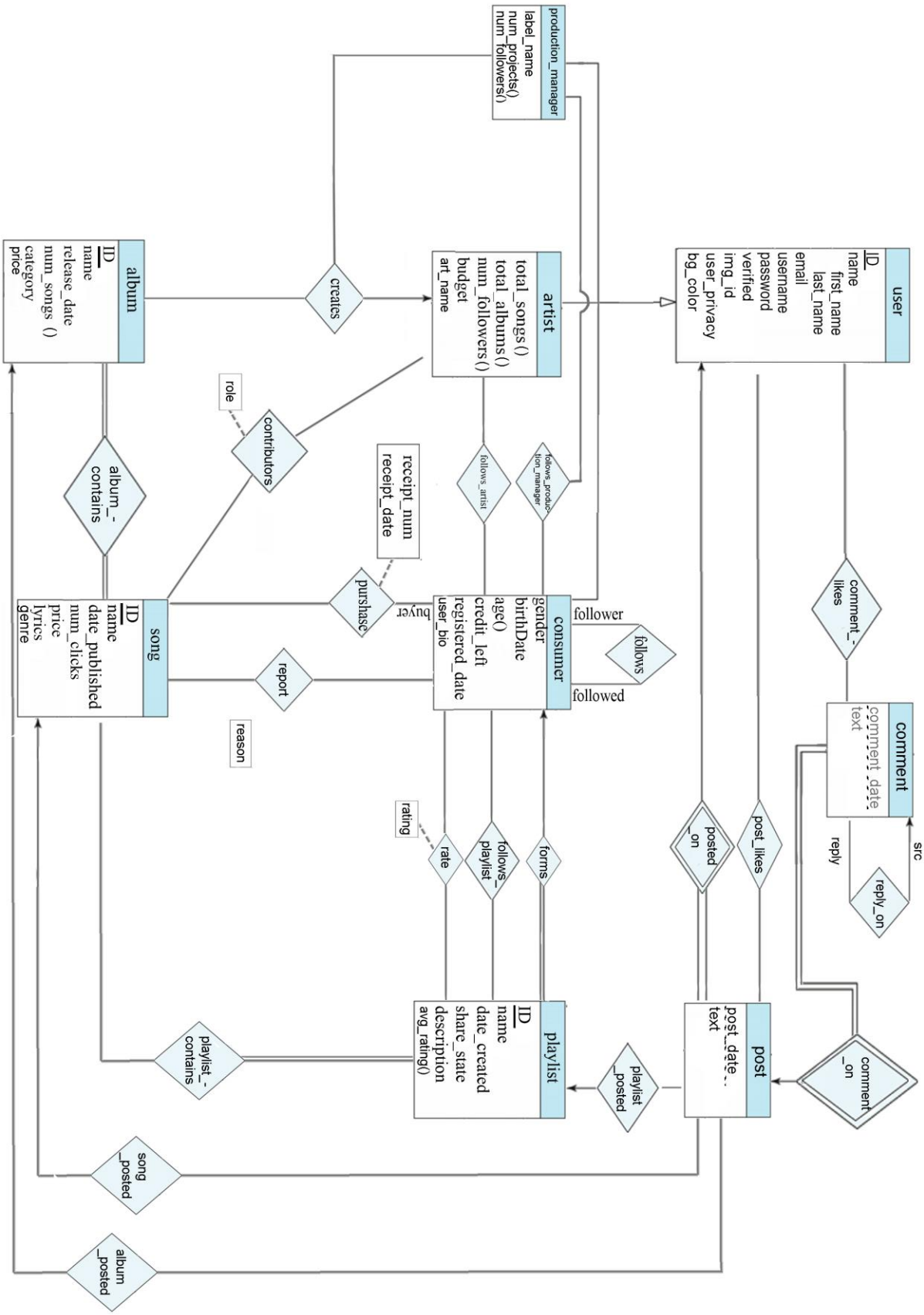

Figure 1-**Revised E/R Model**

2. RELATION SCHEMAS

2.1 User

Relational Model:

user (ID, first_name, last_name, email, username, password, verified, img_id, user_privacy, bg_color)

Functional Dependencies:

ID --> first_name, last_name, email, username, password, verified, img_id, user_privacy, bg_color

username --> ID**,** first_name, last_name, email, password, verified, img_id, user_privacy, bg_color

email --> ID, first_name, last_name, username, password, verified, img_id, user_privacy, bg_color,

Candidate Keys:

{(ID), (username), (email)}

Normal Form:

BCNF

Table Definition:

CREATE TABLE IF **NOT EXISTS** `database`.` **user** ` (
`ID` **INT NOT NULL** AUTO_INCREMENT,
`first_name` **VARCHAR(40),**
`last_name` **VARCHAR(40)**,
`email` **VARCHAR(40) NOT NULL** ,
`username` **VARCHAR(40) NOT NULL** ,
`password` **VARCHAR(40) NOT NULL** ,
`verified` **VARCHAR(3) NOT NULL** ,
`img_id ` **VARCHAR(40),**
`user_privacy` **VARCHAR(8) NOT NULL**,
`bg_color ` **VARCHAR(10),**
PRIMARY KEY (`ID`),
ENGINE = InnoDB

2.2 Artist

artist (ID, budget, art_name)

FK: ID to user

Functional Dependencies:

ID --> budget, art_name

Candidate Keys:

{(ID)}

Normal Form:

BCNF

Table Definition:

NOTE: We do not write total_songs, total_albums and num_followers here for that is calculated. We write the budget as we pay him periodically so we can reset the budget every time he is paid.

```
CREATE TABLE IF NOT EXISTS `database`.` artist` (
`ID` INT NOT NULL ,
`budget` INT NOT NULL,
`art_name` VARCHAR(15),
PRIMARY KEY (`ID`),
CONSTRAINT `ID`
  FOREIGN KEY (`ID`)  REFERENCES user (`ID`))
ENGINE = InnoDB
```

2.3 Consumer

consumer(ID, gender, birth_date, credit_left, registered_date, user_bio)

FK: ID to user

Functional Dependencies:

ID --> gender, birth_date, credit_left, registered_date, user_bio

Candidate Keys:

{(ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` consumer` (
`ID` INT NOT NULL,
`gender` VARCHAR(10),
`credit_left` INT NOT NULL ,
`birth_date ` TIMESTAMP NOT NULL,
`registered_date ` TIMESTAMP NOT NULL,
`user_bio` TINYTEXT ,
PRIMARY KEY (`ID`),
CONSTRAINT `ID`
  FOREIGN KEY (`ID`)  REFERENCES user (`ID`))
ENGINE = InnoDB
```

2.4 Production_manager

production_manager (ID, label_name)

FK: ID to user

Functional Dependencies:

ID --> label_name

Candidate Keys:

{(ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` production_manager ` (
`ID` INT NOT NULL,
`label_name` VARCHAR(20),
PRIMARY KEY (`ID`),
CONSTRAINT `ID`
  FOREIGN KEY (`ID`)  REFERENCES user  (`ID`))
ENGINE = InnoDB
```

2.5 Post

post (ID, post_date, text, playlist_ID, song_ID, album_ID)

FK: ID to user

FK: song_ID to song (ID)

FK: album_ID to album (ID)

FK: playlist_ID to playlist (ID)

Discriminator: post_date

Functional Dependencies:

ID, post_date --> text, playlist_ID, song_ID, album_ID

Candidate Keys:

{(ID, post_date)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.`post ` (
`ID` INT NOT NULL,
`playlist_ID` INT NOT NULL,
`song_ID` INT NOT NULL,
`album_ID` INT NOT NULL,
`post_date` TIMESTAMP NOT NULL,
`text ` TEXT,
PRIMARY KEY (`ID`,`post_date`),
CONSTRAINT `ID`
  FOREIGN KEY (`ID`) REFERENCES user (`ID`)
CONSTRAINT `ID`
  FOREIGN KEY (`song_ID`) REFERENCES song (`ID`)
CONSTRAINT `ID`
  FOREIGN KEY (`album_ID`) REFERENCES album (`ID`))
ENGINE = InnoDB
```

2.6 Comment

comment (ID, post_date, comment_date, text, commenter_ID, replier_ID, reply_date)

FK: ID to user

FK: replier_ID, commenter_ID to user (ID)

FK: post_date to post

FK: reply_date to comment (comment_date)

Discriminator: comment_date, reply_date

Functional Dependencies:

ID, comment_date, post_date, commenter_ID, replier_ID, reply_date --> text

Candidate Keys:

{(ID, comment_date, post_date, commenter_ID, replier_ID, reply_date)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` comment ` (
`ID` INT NOT NULL,
`commenter_ID` INT NOT NULL,
`replier_ID` INT NOT NULL ,
`post_date` TIMESTAMP NOT NULL,
`comment_date` TIMESTAMP NOT NULL,
`reply_date` TIMESTAMP NOT NULL,
`text ` TEXT,
PRIMARY KEY (`ID`, `post_date`,`comment_date`,
`commenter_ID`,`replier_ID`,`reply_date`),
CONSTRAINT `ID`
  FOREIGN KEY (`ID`) REFERENCES user (`ID`)
CONSTRAINT `replier_ID`
  FOREIGN KEY (`replier_ID`) REFERENCES user (`ID`)
CONSTRAINT ` commenter_ID `
  FOREIGN KEY (`commenter_ID `) REFERENCES user (`ID `)
 CONSTRAINT `post_date`
  FOREIGN KEY (`post_date`) REFERENCES post (`post_date`)
CONSTRAINT `reply_date`
  FOREIGN KEY (`reply_date`) REFERENCES comment (`reply_date`))
ENGINE = InnoDB
```

2.7 Likes

2.7.1 Post_likes

post_likes (ID, creator_ID, post_date)

FK: ID to user

FK: creator_ID to user (ID)

FK: post_date to post

Functional Dependencies:

None

Candidate Keys:

{(ID, creator_ID, post_date)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` post_likes ` (
`ID` INT NOT NULL,
`creator_ID` INT NOT NULL,
`post_date` TIMESTAMP NOT NULL,
PRIMARY KEY (`ID`, `post_date`, `creator_ID`),
CONSTRAINT `creator_ID`
  FOREIGN KEY (`creator_ID`) REFERENCES user (`ID`)
CONSTRAINT `ID`
  FOREIGN KEY (`ID`) REFERENCES user (`ID`)
CONSTRAINT `post_date`
  FOREIGN KEY (`post_date`) REFERENCES post (`post_date`))
ENGINE = InnoDB
```

2.7.2 Comment_likes

comment_likes (liker_ID, creator_ID, post_date, commenter_ID, comment_date)

FK: liker_ID, commenter_ID, creator_ID to user (ID)

FK: post_date to post

FK: comment_date to comment

Functional Dependencies:

None

Candidate Keys:

{(liker_ID, creator_ID, post_date, commenter_ID, comment_date)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` comment_likes` (
`liker_ID` INT NOT NULL ,
`comenter_ID` INT NOT NULL ,
`creator_ID` INT NOT NULL,
`post_date` TIMESTAMP NOT NULL,
`comment_date` TIMESTAMP NOT NULL,
PRIMARY KEY (`liker_ID `, `post_date`,`comment_date`, `commenter_ID`,`creator_ID`),
CONSTRAINT `creator_ID`
  FOREIGN KEY (`creastor_ID`) REFERENCES user (`ID`)
CONSTRAINT `replier_ID`
  FOREIGN KEY (`replier_ID`) REFERENCES user (`ID`)
CONSTRAINT `liker_ID`
  FOREIGN KEY (`liker_ID`) REFERENCES user (`ID`)
CONSTRAINT `post_date`
  FOREIGN KEY (`post_date`) REFERENCES post (`post_date`)
 CONSTRAINT `comment_date`
  FOREIGN KEY (`comment_date`) REFERENCES comment (`comment_date`) )
ENGINE = InnoDB
```

2.8 Follows

follows (following_consumer_ID, followed_consumer_ID)

FK: following_consumer, followed_consumer to user (ID)

Functional dependencies:

None

Candidate keys:

{(following_consumer_ID, followed_consumer_ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` follows` (
`following_consumer_ID` INT NOT NULL ,
`followed_consumer_ID` INT NOT NULL,
PRIMARY KEY (`following_consumer _ID `, ` followed_consumer _ID `),
CONSTRAINT ` following_consumer _ID`
  FOREIGN KEY (`following_consumer _ID`)
   REFERENCES user (`ID`)
CONSTRAINT ` followed_consumer _ID`
  FOREIGN KEY (`followed_consumer _ID`)
  REFERENCES user (`ID`) )
ENGINE = InnoDB
```

2.9 Playlist

playlist (playlist_ID, playlist_name, date_created, share_state, description, creator_ID)

FK: creator_ID to user (ID)

Functional Dependencies:

playlist_ID ⟶ playlist_name, date_created, share_state, description, creator_ID

Candidate Keys:

{(playlist_ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` playlist` (
`playlist_ID` INT NOT NULL AUTO_INCREMENT,
`creator_ID` INT NOT NULL,
`playlist_name` VARCHAR(30) NOT NULL,
`share_state` VARCHAR(3) NOT NULL ,
`description ` TINYTEXT,
`date_created` TIMESTAMP NOT NULL,
PRIMARY KEY (`playlist_ID `),
CONSTRAINT `creator_ID`
  FOREIGN KEY (`creastor_ID`)  REFERENCES user (`ID`))
ENGINE = InnoDB
```

2.10Follows

2.10.1Follows_artist

follows_artist (artist_ID, consumer_ID)

FK: artist_ID, consumer_ID to user (ID)

Functional dependencies:

None

Candidate keys:

{(artist_ID, consumer_ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` follows_artist` (
`artist_ID` INT NOT NULL ,
`consumer_ID` INT NOT NULL,
PRIMARY KEY (`artist_ID `, `consumer_ID `),
CONSTRAINT ` artist_ID`
  FOREIGN KEY (`artist_ID`) REFERENCES user (`ID`)
CONSTRAINT ` consumer_ID`
  FOREIGN KEY (`consumer_ID`) REFERENCES user (`ID`)
ENGINE = InnoDB
```

2.10.2 Follows_manager

follows_production_manager (ID, consumer_ID)

FK: ID, consumer_ID to user (ID)

Functional dependencies:

None

Candidate keys:

{(ID, consumer_ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` follows_production_manager ` (
`ID` INT NOT NULL ,
`consumer_ID` INT NOT NULL,
PRIMARY KEY (`ID `, `consumer_ID `),
CONSTRAINT ` ID`
  FOREIGN KEY (`ID`)  REFERENCES user (`ID`)
CONSTRAINT ` consumer _ID`
  FOREIGN KEY (`consumer _ID`)  REFERENCES user (`ID`) )
ENGINE = InnoDB
```

2.10.3 Follows_playlist

follows_playlist (ID, playlist_ID)

FK: playlist_ID to playlist (ID)

FK: ID to user (ID)

Functional dependencies:

None

Candidate keys:

{(ID, playlist_ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` follows_playlist` (
`ID` INT NOT NULL ,
`playlist_ID` INT NOT NULL,
PRIMARY KEY (`ID `, `playlist_ID `),
CONSTRAINT ` ID`
  FOREIGN KEY (`ID`)  REFERENCES user  (`ID`)
CONSTRAINT ` playlist _ID`
  FOREIGN KEY (`playlist _ID`)  REFERENCES playlist (`ID`))
ENGINE = InnoDB
```

2.11 Rate

Rate (ID, playlist_ID, rating)

FK: playlist_ID to playlist (ID)

FK: ID to user (ID)

Functional dependencies:

ID, playlist_ID --> rating

Candidate keys:

{(ID, playlist_ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`. `rate` (
`ID` INT NOT NULL ,
`playlist_ID` INT NOT NULL,
`rating` DOUBLE NOT NULL,
PRIMARY KEY (`ID `, `playlist_ID `),
CONSTRAINT ` ID`
  FOREIGN KEY (`ID`)  REFERENCES user  (`ID`)
CONSTRAINT ` playlist _ID`
  FOREIGN KEY (`playlist _ID`)  REFERENCES playlist (`ID`) )
ENGINE = InnoDB
```

2.12 Album

album (ID, name, release_date, category, price)

Functional dependencies:

ID --> name, release_date, category, price

Candidate keys:

{(ID)}

Normal Form:

BCNF

Table Definition:

CREATE TABLE IF **NOT EXISTS** `database`.` a**lbum**` (
`ID` **INT NOT NULL** AUTO_INCREMENT,
`name` **VARCHAR(30) NOT NULL**
`category` **VARCHAR(20) NOT NULL**
`price` **INT NOT NULL**
`release_date` **TIMESTAMP NOT NULL,**
PRIMARY KEY (`ID `),
ENGINE = InnoDB

2.13 Creates

creates (album_ID, manager_ID, artist_ID)

FK: manager_ID, manager_ID to user (ID)

FK: album_ID to album (ID)

Functional dependencies:

None

Candidate keys:

{(manager_ID, album_ID, artist_ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` creates` (
`album_ID` INT NOT NULL ,
`manager_ID` INT NOT NULL,
`artist_ID` INT NOT NULL,
PRIMARY KEY (`album_ID `, `manager_ID `, ‘artist_ID `),
CONSTRAINT ` album _ID`
  FOREIGN KEY (`album _ID`)  REFERENCES album  (`ID`)
CONSTRAINT ` manager _ID`
  FOREIGN KEY (`manager _ID`)  REFERENCES production_manager (`ID`)
CONSTRAINT ` artist_ID`
  FOREIGN KEY (`artist_ID`)  REFERENCES artist (`ID`) )
ENGINE = InnoDB
```

2.14 Song

song (ID, song_name, date_published, num_clicks, price, lyrics, genre)

Functional dependencies:

ID --> song_name, date_published, num_clicks, price, lyrics, genre

Candidate keys:

{(ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` song` (
`ID` INT NOT NULL AUTO_INCREMENT,
`name` VARCHAR(30) NOT NULL,
`num_clicks` INT NOT NULL ,
`price` INT NOT NULL ,
`lyrics` MEDIUMTEXT,
`genre` VARCHAR(15) NOT NULL
`date_published ` TIMESTAMP NOT NULL,
PRIMARY KEY (`ID`)),
ENGINE = InnoDB
```

2.15 Album_contains

album_contains (album_ID, song_ID)

FK: album_ID to album(ID)

FK: song_ID to song (ID)

Functional dependencies:

None

Candidate keys:

{(song_ID,album_ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` album_contains` (
`album_ID` INT NOT NULL,
`song_ID` INT NOT NULL,
PRIMARY KEY (`song_ID`, `album_ID`)),
ENGINE = InnoDB
```

2.16 Playlist_contains

playlist_contains (song_ID, playlist_ID)

FK: song_ID to song (ID)

FK: playlist_ID to playlist (ID)

Functional dependencies:

None

Candidate keys:

{(song_ID, playlist_ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`. ` playlist_contains ` (
`playlist_ID` INT NOT NULL ,
`song_ID` INT NOT NULL,
PRIMARY KEY (`playlist_ID `, `song_ID `),
 CONSTRAINT ` playlist_ID`
   FOREIGN KEY (`playlist_ID`)  REFERENCES playlist  (`ID`)
CONSTRAINT ` song _ID`
   FOREIGN KEY (`song _ID`)  REFERENCES song (‘ID`) )
ENGINE = InnoDB
```

2.17 Contributors

contributors (ID, song_ID, role)

FK: ID to user (ID)

FK: song_ID to song (ID)

Functional dependencies:

ID, song_ID --> role

Candidate keys:

{(ID, song_ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`. `contributors` (
`ID` INT NOT NULL ,
`song_ID` INT NOT NULL,
`role` VARCHAR(15) NOT NULL,
PRIMARY KEY (`ID `, `song_ID `),
CONSTRAINT ` ID`
  FOREIGN KEY (`ID`)  REFERENCES user (`ID`)
CONSTRAINT ` song _ID`
  FOREIGN KEY (`song _ID`)  REFERENCES song (`ID`))
ENGINE = InnoDB
```

2.18 Purchase

purchase (ID, song_ID, receipt_num, receipt_date)

FK: ID to user (ID)

FK: song_ID to song (ID)

Functional dependencies:

ID, song_ID --> receipt_num, receipt_date

Candidate keys:

{(song_ID, ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` purchase` (
`ID` INT NOT NULL ,
`song_ID` INT NOT NULL,
`receipt_num` LONG NOT NULL,
`receipt_date` TIMESTAMP NOT NULL,
PRIMARY KEY (`ID `, `song_ID `),
CONSTRAINT `ID`
  FOREIGN KEY (`ID`) REFERENCES user (`ID`)
CONSTRAINT `song_ID`
  FOREIGN KEY (`song_ID`) REFERENCES song (`ID`) )
ENGINE = InnoDB
```

2.19 Report

Report (ID, song_ID, reason)

FK: ID to user (ID)

FK: song_ID to song (ID)

Functional dependencies:

ID, song_ID --> reason

Candidate keys:

{(song_ID, ID)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE IF NOT EXISTS `database`.` report` (
`ID` INT NOT NULL ,
`song_ID` INT NOT NULL,
`reason`  VARCHAR(30) NOT NULL
PRIMARY KEY (`ID `, `song_ID `),
CONSTRAINT `ID`
  FOREIGN KEY (`ID`)  REFERENCES user (`ID`)
CONSTRAINT `song_ID`
  FOREIGN KEY (`song_ID`)  REFERENCES song (`ID`)
ENGINE = InnoDB
```

3.0 FUNCTIONAL COMPONENTS

The relations that we have provided in the previous section of this report are in Boyce-Codd Normal Form (BCNF) therefore our relations do not need normalization or decomposition.

3.1 Use Cases

3.1.1 Consumer

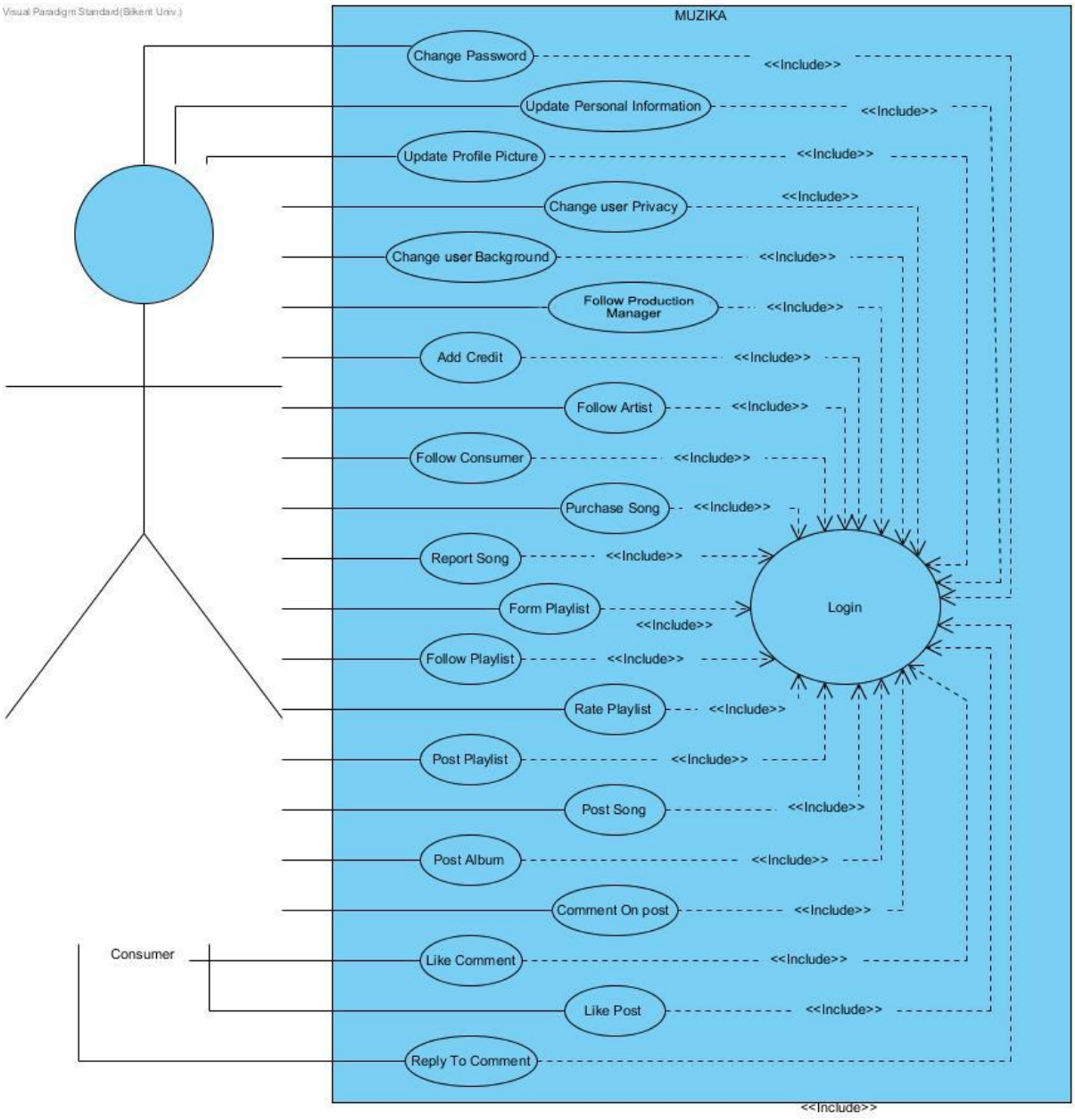

Figure 2-**Consumer Use Case**

Change Password: The consumer has the possibility to change the password whenever they want or need to.

Update Personal Information: The consumers can add their biography and also they can change this information whenever they want to.

Change User Privacy: Consumers can change their privacy so that they can allow only their followers to view what they post or make the privacy public so that everyone can see the posts.

Change Background Color: Consumers can change their profile background color according to their preferences.

Add Credit: Consumers need credit to purchase songs and albums. Therefore in order to have sufficient credits they need to increase their credit from time to time by paying in the system. (Payment method is not covered in our project)

Follow Artist: Consumers can follow artists so that they can see their new posts.

Follow Consumers: Consumers can follow other consumers in order to see their posts and share or rate their playlists.

Purchase Song: Consumers can purchase songs so that they can listen to the full version of the song. If they don't purchase it, they will only be able to listen to 29 seconds of a song.

Report Song: Consumers can also report a song if they don't like it. If the user reports a song, he/she will not be able to listen to that song again.

Form Playlist: The consumers can form playlists so that they can listen to a sequence of songs they have chosen. In the playlist they can only add songs that they have already purchased.

Rate Playlist: If consumers have followed other consumers, they can rate the playlists that they have shared.

Post Playlist: Consumers can post a playlist that they have created on their wall.

Post Song: Consumers can post a song that they have purchased on their wall.

Post Album: If consumers have purchased a full album they can post it on their wall.

Comment on Post: Consumers can comment on their post on their friend's posts.

Like Post: Consumers can like their own post or a post of another consumer or artist.

Like Comment: Consumers can like their own comment or the comment of another user.

Reply To Comment: Consumers can reply to their own comment or to the comment of another user.

3.1.2 Artist

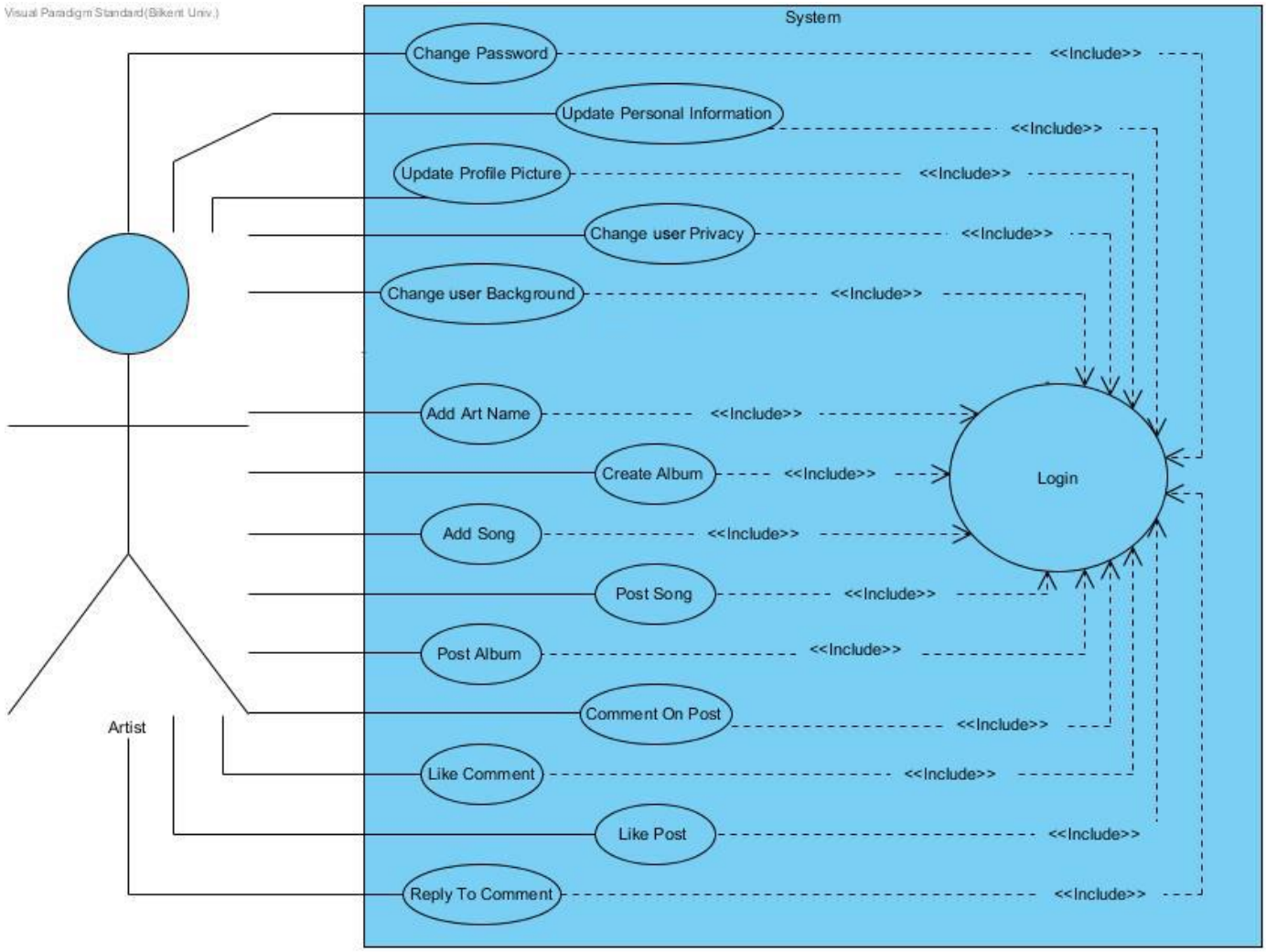

Figure 3-**Artist Use Case**

Change Password: The artist has the possibility to change the password whenever they want or need to.

Update Personal Information: The artists can add their biography and also they can change this information whenever they want to.

Change User Privacy: Artists can change their privacy so that they can allow only their followers to view their posts or make the privacy public so that everyone can see them.

Change Background Color: Artists can change their profile background color according to their preferences.

Add Art Name: Artists can add an art name such as their nick name or the name by which they are known to the audience.

Create Album: Artist can create albums and then add songs to their albums.

Add Song: Artists can add songs to an album or as singles. If songs are added as singles they will be automatically added in a single album.

Post Song: Artist can post their songs on their wall so that people who follow them can listen to them in a full version if they have purchased it or in a demo version if they haven't purchased it.

Post Album: Artists can also post their album will all the songs contained in it.

Comment on Post: Artist can make a comment on the post that they have posted on their wall.

Like Post: Artist can like their own post.

Like Comment: Artists can like a comment on their post.

Reply to Comment: Artist can reply to a comment that their followers have done on their post.

3.1.3 Production Manager

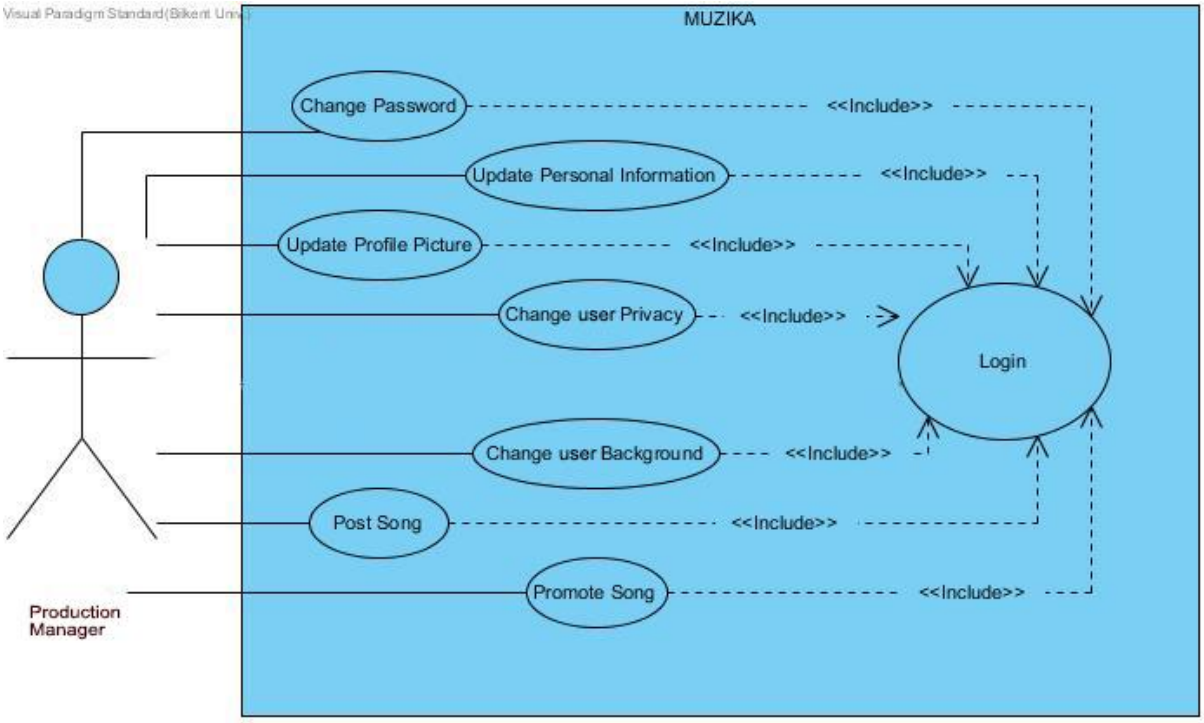

Figure 4-**Production manager Use Case**

Change Password: The production manager has the possibility to change the password whenever they want or need to.

Update Personal Information: The production manager can add their biography and also they can change this information whenever they want to.

Change Privacy: Production Managers can change their privacy so that they can allow only their followers to view their posts or make the privacy public so that everyone can see them.

Change Background Color: Production Managers can change their profile background color according to their preferences.

Post Song: Production Managers can post the songs they have produced on their wall so that people who follow them can listen to them in a full version if they have purchased it or in a demo version if they haven't purchased it.

3.2 Algorithms

3.2.1 Logical Requirements

The main logical errors are caused by the usage of date and time types. Therefore in our system we will be careful in checking boundary dates so that we prevent and avoid logical errors.

In our system we are using **date** attribute as descriptors of **post** and **comment**. Therefore we should be careful in checking dates since they will be primary keys for the relation between post and user and also between comment and user.

3.2.2 Entity Related Algorithms

MUZIKA is a system which provides the possibility to make posts and comments. Furthermore, we also have provided the opportunity to reply to a comment, to make a comment on a post, like a post or comment, etc. This means that whenever a user deletes a post or comment, the tuples which contain the primary key of the deleted entity should all be removed from the other relations in order to avoid other logical problems. For example, if a post is deleted, the tuples which contain the same **user ID, date** in **likes_post** should also be removed from the relations.

3.3 Data Structures

In our system, the relations we have created contain data types such as NUMERIC TYPES, STRING, DATE, TIME TYPES.

INT is the NUMERIC TYPE we have used which is needed for the IDs, number of likes, number of rates.

VARCHAR is the STRING TYPE we have used in order to describe all the attributes composed of alphanumeric types.

TIMESTAMP is used for all attributes which denote date and time.

MEDIUMTEXT is used for the song lyrics and TINYTEXT is used for user bio and playlist description to set an upper limit for these attributes.

TEXT is used for posts and comments to allow the users to be more interactive with longer comments

4. USER INTERFACE AND SQL STATEMENTS

4.1 Login Screen

Figure 5-Login Screen

Inputs: @email, @password

Process: Users may login to the platform by entering their username and passwords

SQL Statements:

SELECT ID
FROM user
WHERE username = @email **AND** password = @password

4.2 Sign Up Screen

4.2.1 Consumer

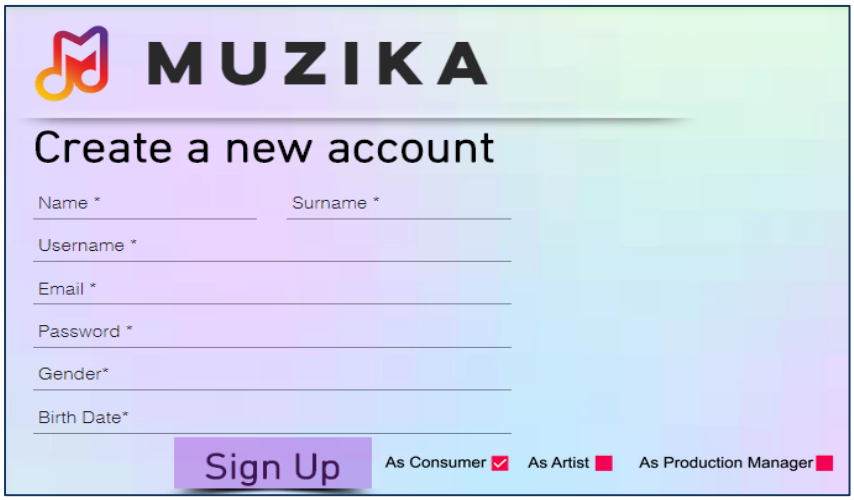

Figure 6-Consumer Sign Up Screen

Inputs: @first_name, @last_name , @username, @email, @password, @gender, @birthDate

Process: Users wishing to sign up as consumers have to check the box 'As Consumers' as shown in the picture above. The corresponding form, as shown in the picture will prompt the Consumer to enter their information and then sign up through the 'Sign Up' button. There cannot be two consumers with the same username, or the same email address.

(**Note:** Several values were omitted from the input list: **@ID**, **@verified,** **@img_id**, **@user_privacy**, **@bg_color**, **@user_bio**, **@credit_left**, **@registered_date** because they will be generated in default hardcoded values. The @ID does not need to be in the list because it will be set to auto increment, thus supplying NULL for ID, will let the system use auto-increment on its value instead)

SQL Statements:
// Check if username or email already exists with 5.1
SELECT check_user_exists(@email, @username**)**

INSERT INTO user
VALUES (NULL , @first_name, @last_name, @email, @username, @password, @verified: **no** , @img_id: **0**, user_privacy: **public**, @bg_color: **ffffff**)

SET @consumerID = **SELECT** ID **FROM** user **WHERE email** = @email

// after user clicks the verification link in their email, account is verified, we set the value to 'yes'
UPDATE user
SET verified = 'yes'
WHERE ID = @consumerID

// delete the user if not verified within 24 hours

DELETE FROM user
WHERE ID = @consumerID

// insert user information into consumer table if verified
INSERT INTO consumer
VALUES (@consumerID, @user_bio: **I am new here**, @gender, @birthdate, @credit_left: **5000,** NOW())

4.2.2 Artist

Figure 7-Artist Sign Up Screen

Inputs: @first_name, @last_name , @username, @email, @password, @art_name

Process: Users wishing to sign up as artists have to check the box 'As Artist' as shown in the picture above from the Sign Up Section. The corresponding form, as shown in the picture will prompt the artist to enter their information and then sign up through the 'Sign Up' button. An artist cannot create an account that has the same art username as an already existing user. Also they are not allowed to use an already existing email address

(**Note:** @ID was left out of the list of inputs because it will be set to auto_increment)

SQL Statements:
// check if artist exists (defined in section 5.2)
SELECT check_artist_exists(@art_name, @email **)**

INSERT INTO user
VALUES (**NULL**, @first_name, @last_name, @email, @username, @password, @verified: **no** , @img_id: **0**, user_privacy: **public**, @bg_color: **ffffff**)

SET @artistID = **SELECT** ID **FROM** user **WHERE** email = @email

INSERT INTO artist
VALUES(@artistID , @budget: 2500, @art_name)

4.2.3 Production Manager

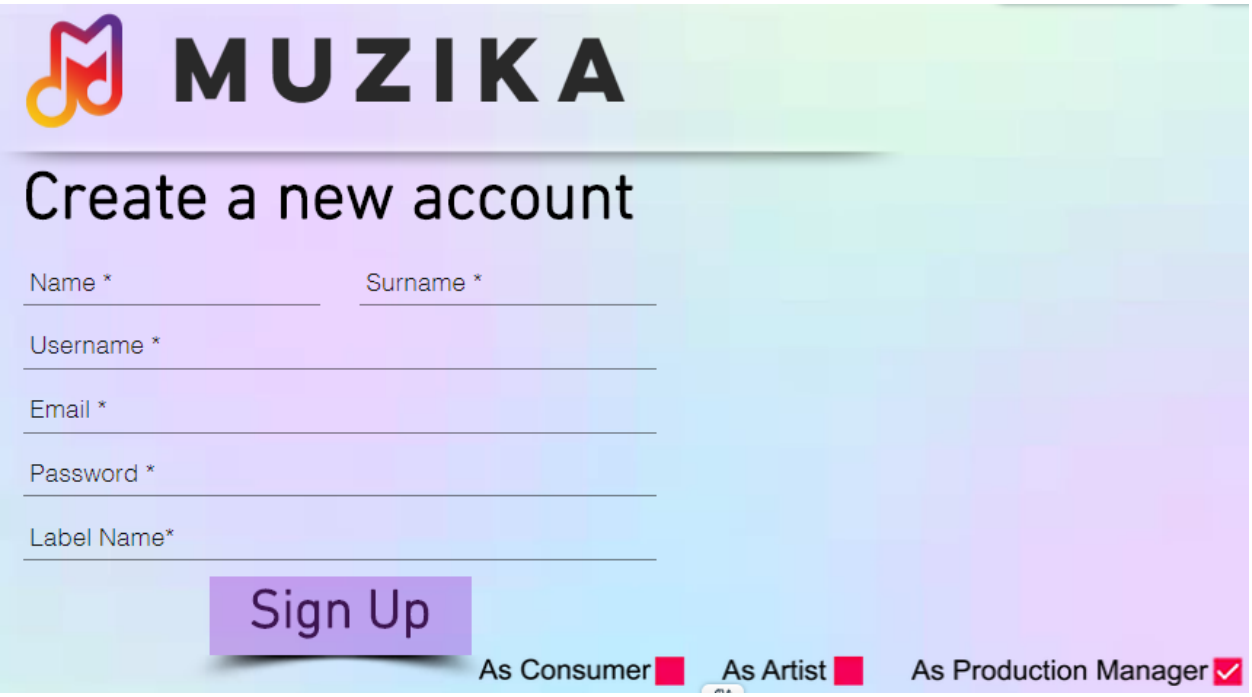

Figure 8-Production Manager Sign Up Screen

Inputs: @first_name, @last_name , @username, @email, @password, @label_name

Process: Users wishing to sign up as production managers have to check the box 'As Production Manager' as shown in the picture above from the Sign Up Section. The corresponding form, as shown in the picture will prompt the artist to enter their information and then sign up through the 'Sign Up' button. Users may not sign up with an already existing username or email

(Note: @ID was left out of the list of inputs because it will be set to auto_increment)

SQL Statements:
// check if user already exists (defined in section 5.1)
SELECT check_user_exists(@email, @username**)**

INSERT INTO user
VALUES (**NULL**, @first_name, @last_name, @email, @username, @password, @verified: **no** , @img_id: **0**, user_privacy: **public**', @bg_color: **ffffff**)

SET @managerID = **SELECT** ID **FROM** user **WHERE email** = @email

INSERT INTO production_manager

VALUES (@managerID, @label_name)

4.3 Profile Edit

4.3.1 Consumer

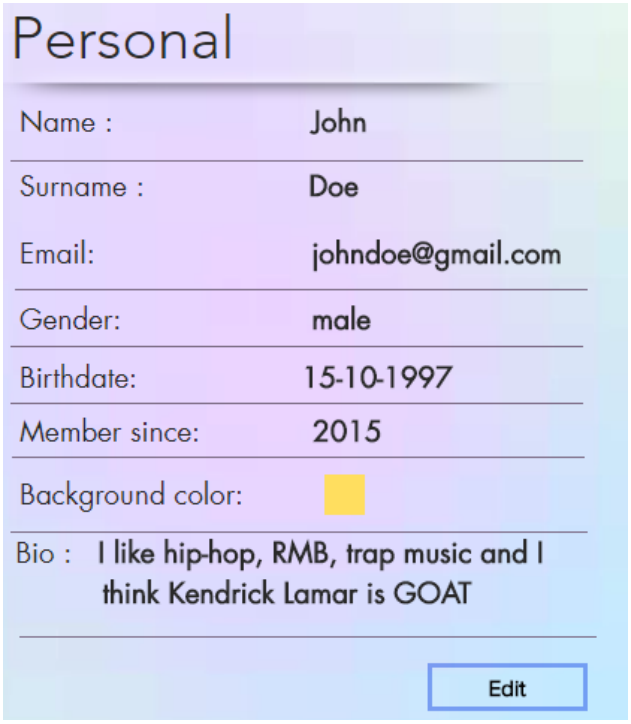

Figure 9-Edit Consumer Profile

Inputs: @first_name, @last_name , @email, @gender, @ birthDate, @user_bio, @bg_color

Process: Consumers may be able to edit/alter their personal information shown in their profile through the edit button. They may be able to change their name, surname, gender, birthdate, background color and bio. However, users can not change the email to one that is already registered on the platform.

SQL Statements:

// Update data, **check_user_email** is defined in section 5.3
UPDATE user, consumer
SET user .first_name = @first_name,
user .last_name=@last_name,
user .email = @email,
user.bg_color = @bg_color,
consumer. gender=@gender,
consumer. birthDate=@birthDate,
consumer .user_bio = @user_bio
WHERE ID = @ID **AND check_user_email(** @ID , @email) = 0

4.3.2 Artist

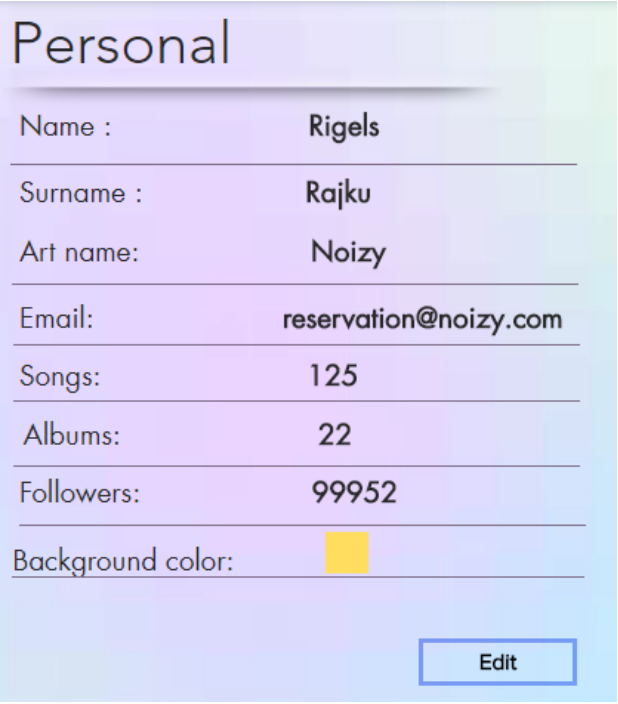

Figure 10-Edit artist profile

Inputs: @first_name, @last_name , @art_name, @email , @bg_color

Process: Artists may be able to change their personal profile information as well, such as name, surname, art name, email address and background color. However, users cannot edit their email to an already existing email address.

SQL Statements:
// update the data using **check_artist_exist_update** defined in chapter 5.4
UPDATE user, artist
SET user .first_name = @first_name, user. last_name=@last_name, user .email=@email, artist.art_name = @art_name
WHERE ID = @ID **AND check_artist_exist_update(** @ID, @artist_name, @email**)** = 0

4.3.3 Production Manager

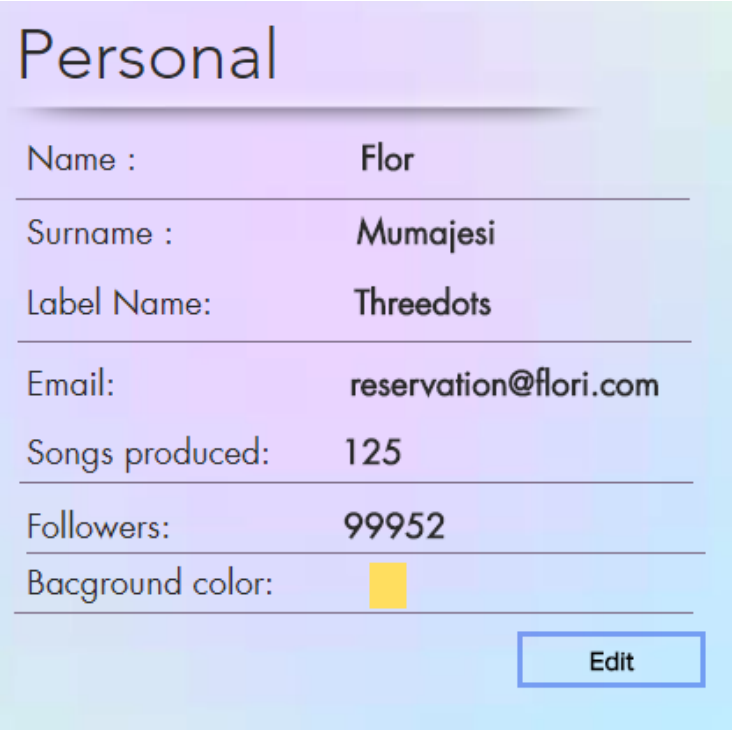

Figure 11-Edit production manager profile

Inputs: @first_name, @last_name , @label_name = , @email , @bg_color

Process: Managers may be able to change their personal profile information as well, such as name, surname, label name, email address and background color. A manager cannot edit his email to an email that already exists.

SQL Statements:

// update the data using **check_manager_exists_update** defined in section 5.5

UPDATE user, production_manager
SET user .first_name = @first_name,
user.last_name = @last_name,
user.email = @email,
production_manager.label_name = @label_name
WHERE ID = @ID **AND check_manager_exists_update(**@ID, @label_name, @email **) =** 0

4.4 Album publishing

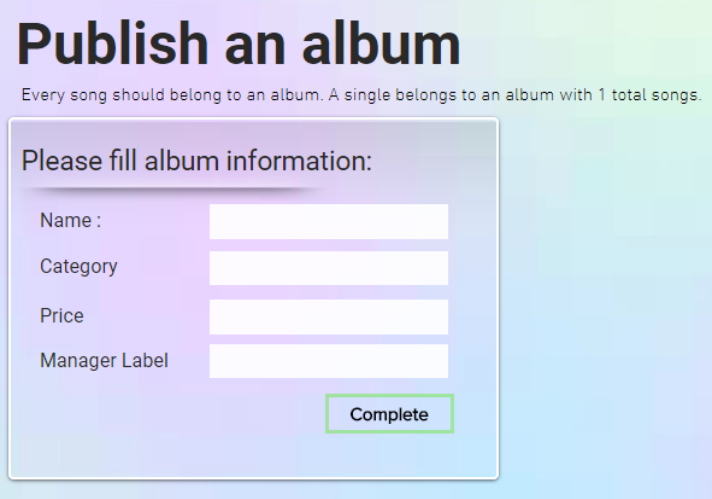

Figure 12-Fill album detail

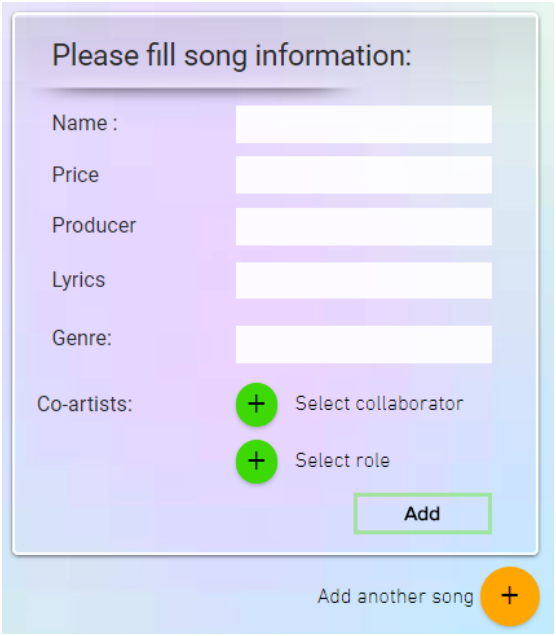

Figure 13-Fill song information

Inputs: @artistID, @production_manager_label_name, @album_name, @category, @album_price, @song_name, @song_price, @song_producer, @song_lyrics, @song_collaborator, @contributor_art_name, @contributor_role , @genre

Process: Artists may publish an album by filling in the information shown in the pictures above. After pressing 'Complete' , the artist has to fill in the information for at least one song to be included in the album. Collaborators to the song are added as well. The Manager label will function as a drop down menu that changes on user input by running a query to get all label names.

SQL Statements

```
//hold the production manager ID corresponding to user given inputs-name and surname- into a variable
SET @managerID  =
SELECT ID
FROM production_manager
WHERE label_name= @label_name

//inserts a new album into the table, NOW() corresponds to the current date in the time of insertion
INSERT INTO album
VALUES( NULL, @album_name, NOW(), @category, @album_price)

//holds  the last inserted albums ID  into a variable, will be needed further
SET @albumID = LAST_INSERT_ID()

//insert a new song in the table, initialliy number of clicks is 0
INSERT INTO song
VALUES( NULL, @song_name, NOW() , @num_clicks = 0, @song_price, @lyrics, @genre)
// after new song is inserted, save its ID into a variable, will be needed further
```

SET @songID = LAST_INSERT_ID()

// insert corresponding ID's of the entities involved in the ternary relationship of the current transaction (defined in earlier steps)
INSERT INTO creates
VALUES (@albumID , @managerID **,** @artistID**)**

//insert ID's of both album and song (defined in earlier steps) to define which songs belong to which album
INSERT INTO album_contains
VALUES(@albumID, @songID)

//return the ID of the collaborator, by getting users input, when pressing the respective art name on the '+' button
SET @collaboratorID =
SELECT ID
FROM artist
WHERE art_name = @contributor_art_name

// insert collaborator ID and respective song ID(defined earlier) into the contributors table. In one iteration only one contributor can be added, so for N contributors, N iterations of this query are needed
INSERT INTO contributors
VALUES (@collaborator_ID, @songID, @contributor_role)

4.5 Creating a playlist

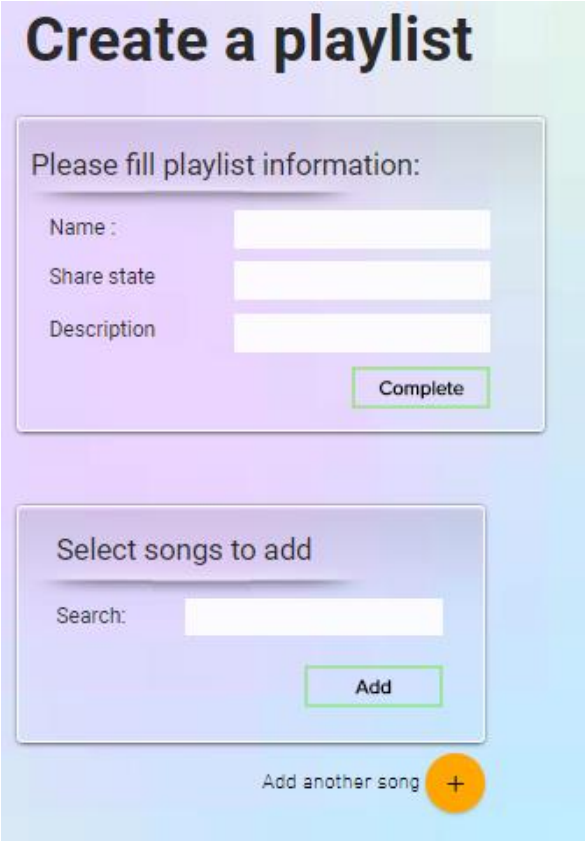

Figure 14-Create Playlist

Input: @playlistName, @shareState, @description, @songName

Process: Consumers may create a new playlist by filling in the information of the playlist and selecting at least one song to add. The search text box will function as an autocomplete drop down which shows only songs that the user has purchased because the user cannot make a playlist of songs he doesn't own

SQL Statements:
// used to obtain all songs that the user owns (user_songs is a view from section **5.6**)
SELECT *
FROM user_songs

//create the playlist
INSERT INTO playlist
VALUES (NULL, @playlistName, NOW(), @share_state, @description)

// after new playlist is inserted, save its ID into a variable, will be needed further
SET @playlistID = LAST_INSERT_ID()

// for all songs that the user selects
INSERT INTO playlist_contains_song
VALUES(@songID, @playlistID)

4.6 Searching the database, for songs, albums, artists, playlists, users

Inputs: @search_term_case_insensitive (*used by all subsections below)

Process: Users can search for information in the search section by entering the search term and pressing the button 'Search'. This search will show information on all songs, albums, artists, playlists, consumers, producers who are related to that search term. **@search_term_case_insensitive** is inherited in all of the subsections below.

4.6.1 Displaying search results for songs and buying them

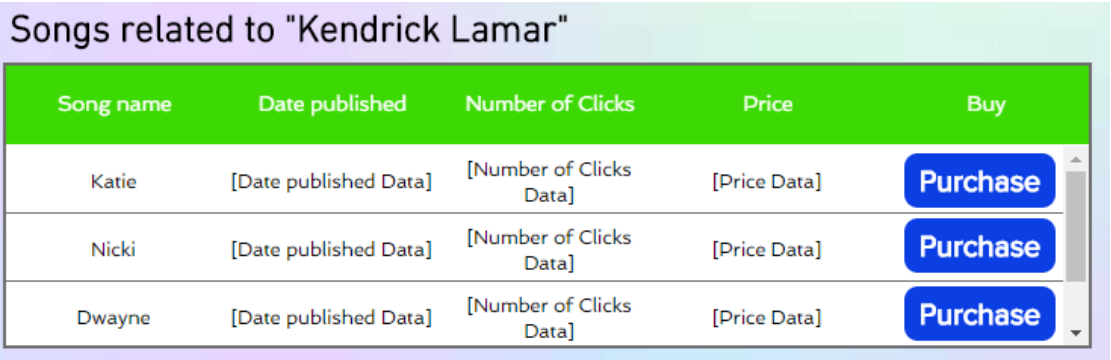

Figure 15-Display Song Results

Inputs: @songID

Process: The song search results are shown and the user may choose to purchase a song. Users can not purchase a song already purchased, since the option will not be given.

(Note: @songID is fetched through a select statement and the corresponding row index on the pressed 'Purchase' button; @receipt_num is randomly generated by the system**)**

SQL Statements

//return all tuples having a substring of the given search input
SELECT (ID, name, date_published, num_clicks, price)
FROM song
WHERE name **LIKE '%**@search_term_case_insensitive **%'**

//get all song id's of already purchased songs to decide whether to show 'Purchase ' button or not
// (user_songs is a view from section **5.6**)
SELECT song_ID
FROM user_songs

// Decrement the consumers credit_left(received from the first SQL query)
UPDATE consumer
SET credit_left = credit_left -@price
WHERE ID = @logged_in_consumerID

// increment aritsts budget
UPDATE artist
SET budget = budget + @price

// add the song to users purchased songs
INSERT INTO purchased
VALUES (@logged_in_consumerID, @songID, NULL, NOW() **)**

4.6.2 Displaying search results for artists

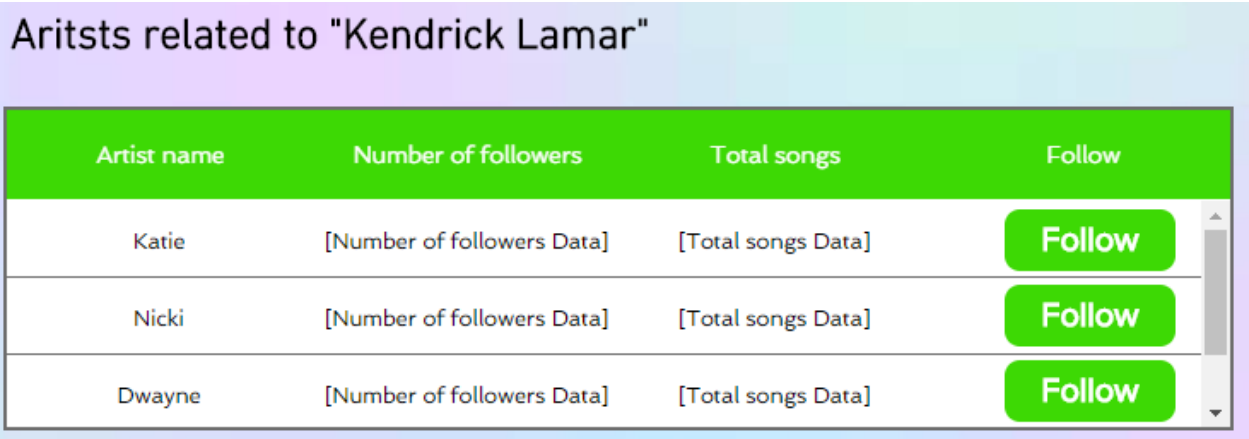

Figure 16-Display Artists related to the queried Artist

Inputs: @artistID

Process: The artist search results are shown and the user may choose to follow an artist. Users can not follow an artist already followed. The follow button will not be showed in the latter case.

SQL Statements:

//return all tuples having a substring of the given search input
SELECT (ID, art_name, num_followers , total_songs)
FROM artist
WHERE art_name **LIKE '%@**search_term_case_insensitive **%'**

// first check to see if consumer already follows the artist using view defined in **5.7**
SELECT *
FROM user_follows_artist
// add the artist into the user's list of followers
INSERT INTO follows_artist
VALUES (@artistID , @logged_in_consumerID**)**

4.6.3 Displaying search results for albums

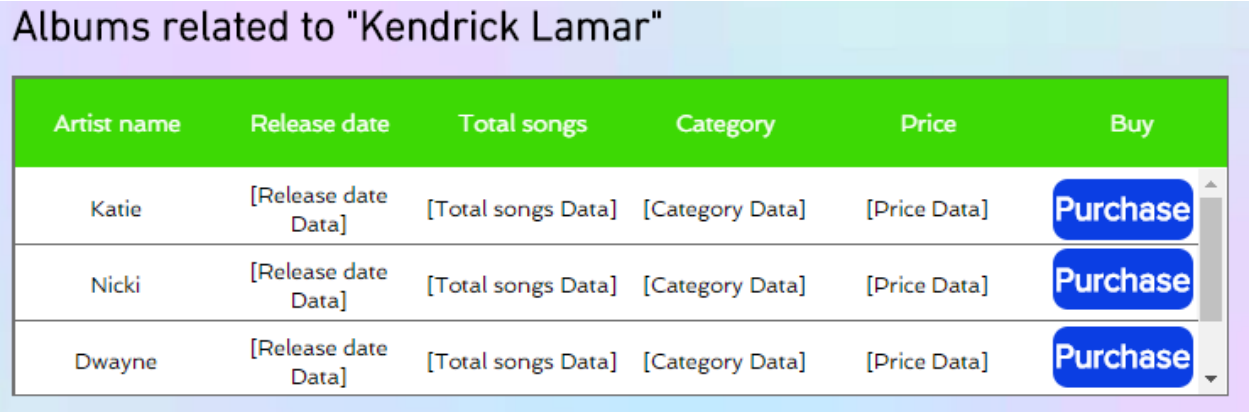

Figure 17-Display albums related to the queried Artist

Inputs: @albumID

Process: The album search results are shown and the user may choose to purchase an album. Users can not purchase an album already purchased. Only consumers may purchase an album

SQL Statements:

//return all tuples having a substring of the given search input
SELECT (ID, name, release_date, num_songs, category, price)
FROM album
WHERE name **LIKE '%@**search_term_case_insensitive **%'**

// check to see if user already purchased the album with using view defined in section **5.8**
SELECT *
FROM user_albums

//thirdly, return all songs in the selected album to be bought
SELECT songID
FROM contains
WHERE albumID = @albumID

//add all songs corresponding to the selected album to the purchased songs table, @receipt_num is auto-incremented
INSERT INTO purchase
VALUES (@logged_in_consumerID, @songID, NULL, NOW() **)**

4.6.4 Displaying search results for playlists

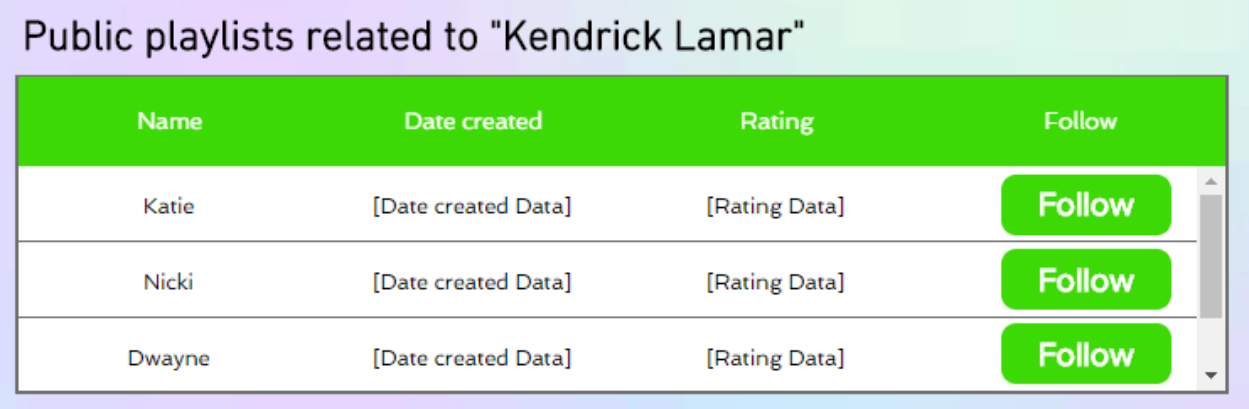

Figure 18-Display playlists related to the queried name

Inputs: @playlistID

Process: The playlist search results are shown and the user may choose to follow a playlist. Users can not follow a playlist already followed. Only consumers may follow a playlist.

SQL Statements:

//return all tuples having a substring of the given search input
SELECT (ID, name, date_created, calc_rating(ID))
FROM playlist
WHERE name **LIKE '%@**search_term_case_insensitive **%'**

// check if user already follows playlist, user_followed_playlists is a view defined in section **5.9**
SELECT *
FROM user_followed_playlists

// insert playlist into the following_playlist table
INSERT INTO follows_playlist
VALUES (@logged_in_consumerID, @playlistID **)**

4.6.5 Displaying search results for consumers

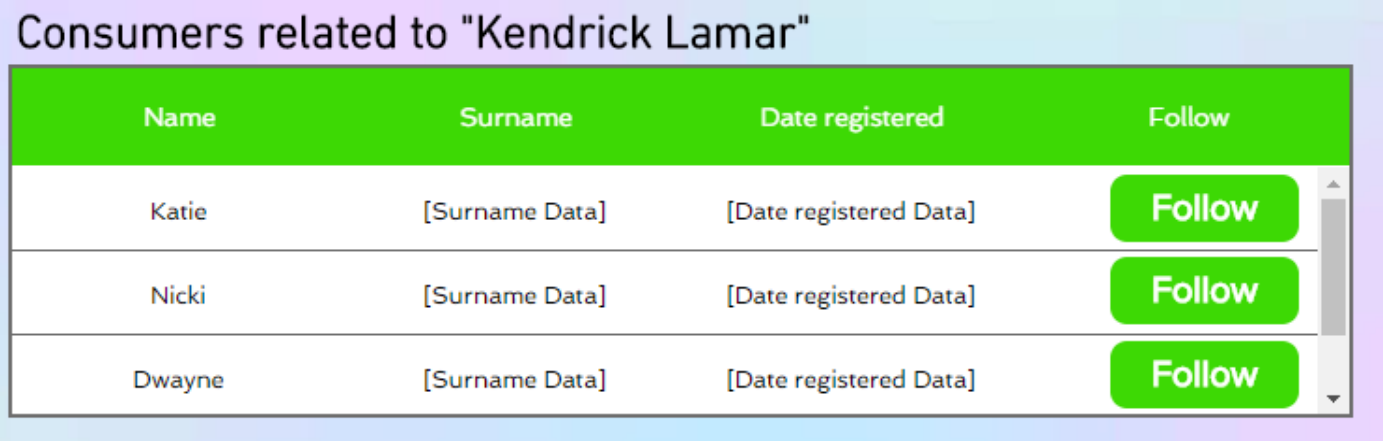

Figure 19-Display Artists related to the queried name

Inputs: @selected_consumerID
Process: The consumer search results are shown and the user may choose to follow a consumer. Users can not follow a consumer already followed. Only consumers may follow a consumer.

SQL Statements:

//return all tuples having a substring of the given search input
SELECT (ID,first_name, last_name, registered_date)
FROM user
WHERE first_name **LIKE '%@**search_term_case_insensitive **%' OR**
last_name **LIKE '%@**search_term_case_insensitive **%'**

// check if user already follows consumer, consumer_follows_consumer defined in section **5.10**
SELECT *
FROM consumer_follows_consumer

// insert @consumerID into userID's *following* table, @consumerID represents the selected consumer on the table given in the picture

INSERT INTO follows
VALUES (@logged_in_consumerID, @selected_consumerID)

4.6.6 Displaying search results for production manager

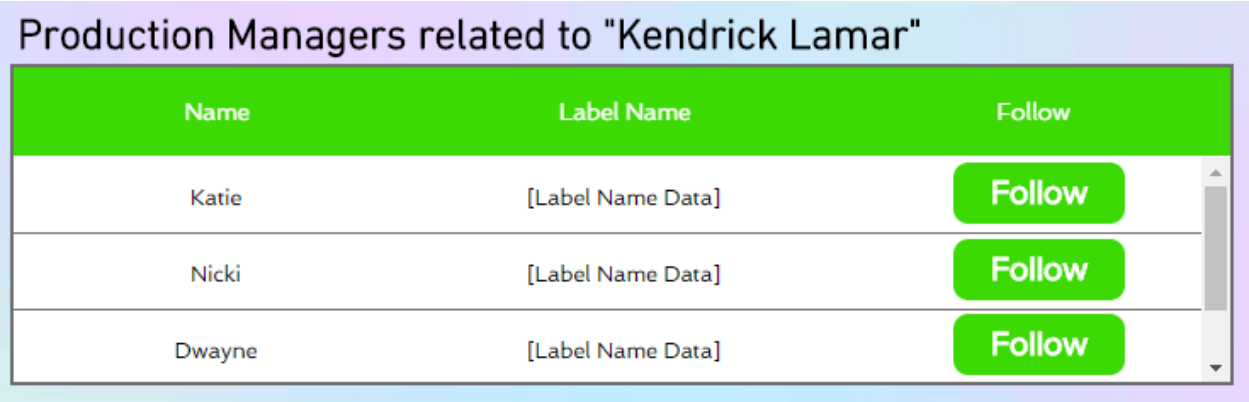

Figure 20-Display managers related to the queried name

Inputs: @managerID

Process: The production manager search results are shown and the consumer may choose to follow a manager. Users can not follow a manager already followed. Only a consumer may follow a manager.

SQL Statements:

//return all tuples having a substring of the given search input
SELECT (ID, first_name_ label_name)
FROM user **NATURAL JOIN** production_manager
WHERE label_name **LIKE '%@**search_term_case_insensitive **%'**

// check if user already follows consumer, user_follows_manager defined in section **5.11**
SELECT *
FROM user_follows_manager

// insert followed manager
INSERT INTO follows_production_manager
VALUES (@logged_in_consumerID, @selected_consumerID)

4.7 Posting

4.7.1 Posting a status

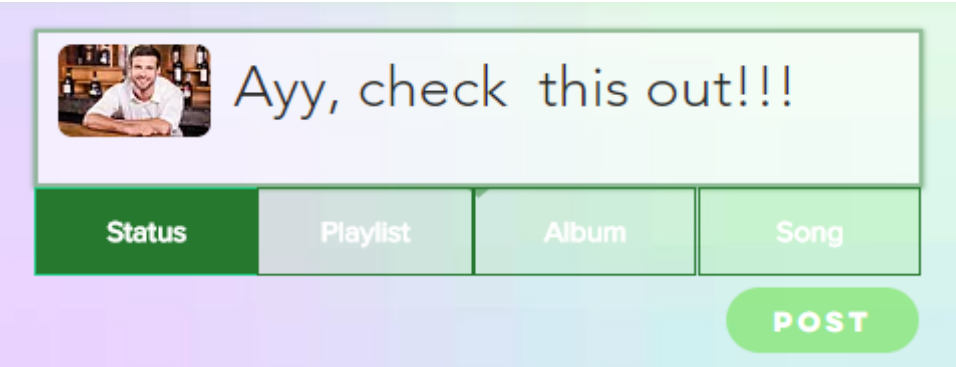

Figure 21-Select Status to create a status post

Input: @status_text

Process: Users may post a status by selecting the status button, filling in the text field and then pressing 'Post'.

SQL Statements:

INSERT INTO post
VALUES(@logged_in_userID, NOW(), @status_text, -1, -1, -1)

4.7.2 Posting a Playlist

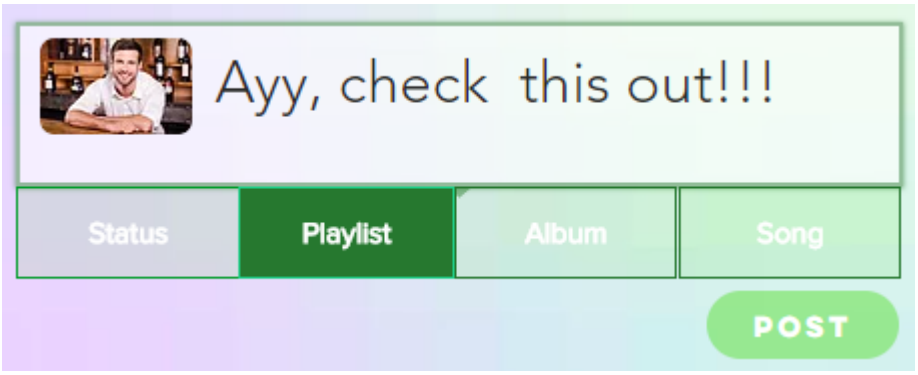

Figure 22-Select Playlist to be able to share a playlist

Input: @status_text, @playlistID

Process: Users may post a playlist adjacent to some description. When selected, the 'Playlist' button will show a list of all the playlists the user has. The user will select a playlist from that window and after that will have to press 'Post' to successfully post the playlist. Users may follow any playlist that is visible to them.

SQL Statements:

//retrieve all playlists ID belonging to the current user

SELECT *
FROM playlists
WHERE share_state = ‘public’

INSERT INTO post
VALUES(@ logged_in_userID, NOW(), @status_text, @playlistID, -1,-1)

4.7.3 Posting an Album

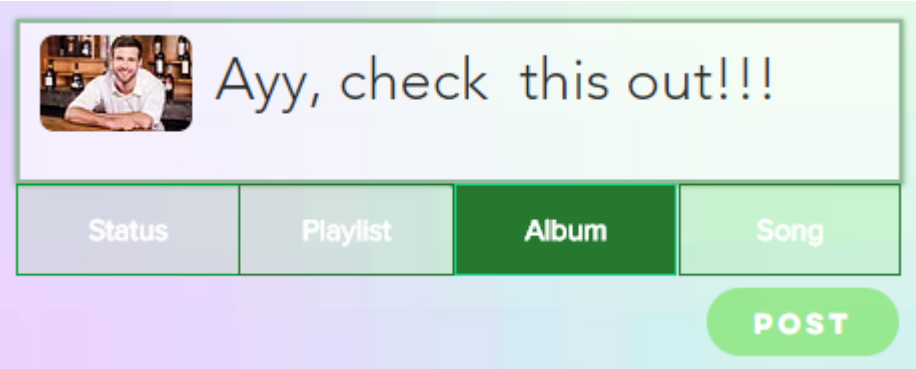

Figure 23-Select Album to create a post with album

Input: @status_text, @albumID

Process: Users may post an album adjacent to some description. When selected, the ‘Album’ button will show a list of all the albums the user owns. The user will select an album from that window and after that will have to press ‘Post’ to successfully post the album.

SQL Statements:

//retrieve all album IDs belonging to the current user, **user_albums** defined in section 5.8
SELECT *
FROM user_albums

INSERT INTO post
VALUES(@logged_in_userID, NOW(), @status_text, -1, @albumID,-1)

4.7.4 Posting a Song

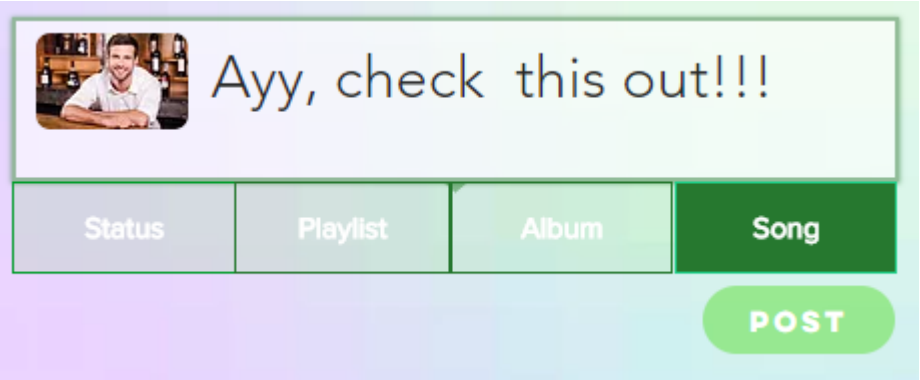

Figure 24-Select Song to create a post with a song

Input: @status_text, @songID

Process: Users may post a song adjacent to some description. When selected, the 'Song' button will show a list of all the songs that the user has purchased. The user will select a song from that drop down box and after that will have to press 'Post' to successfully post the song.

SQL Statements:

//retrieve all song IDs belonging to the current user, **user_songs** is defined in section **5.6**
SELECT *
FROM user_songs

INSERT INTO post
VALUES(@logged_in_userID, NOW(), @status_text, -1, -1, @songID)

4.8 Making a comment, like, reply

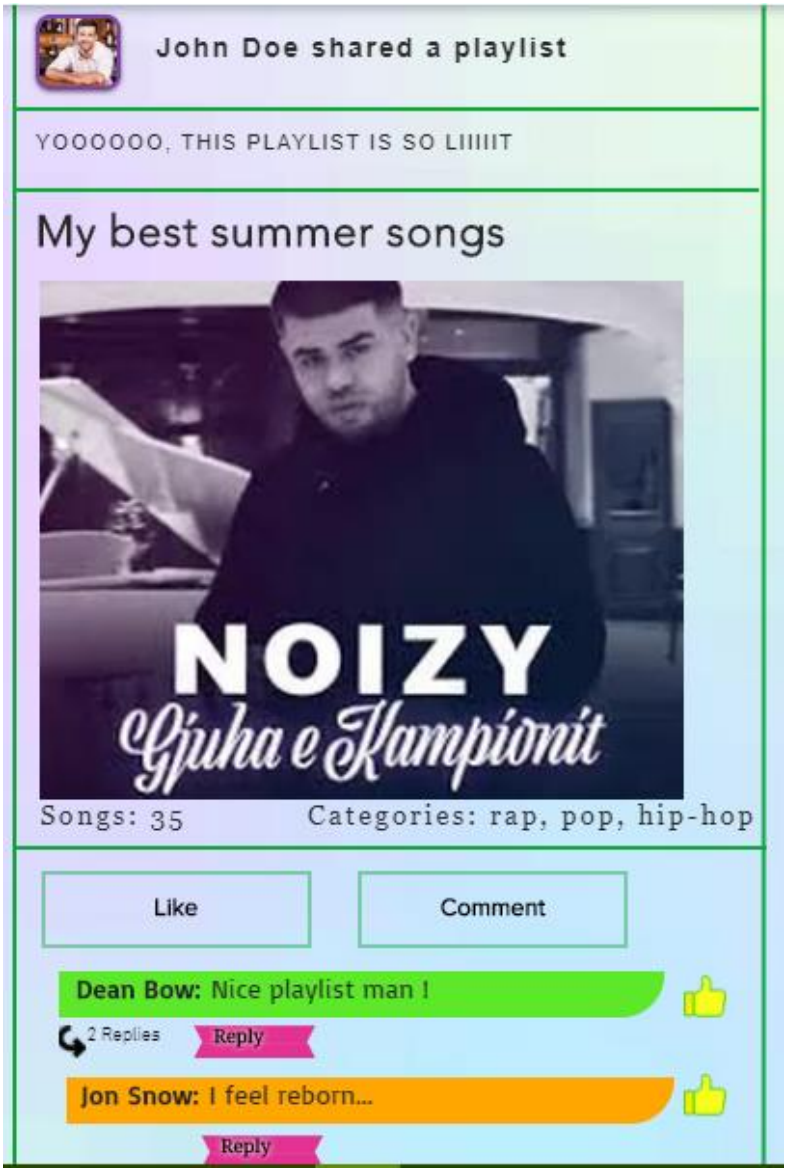

Figure 25-Comments on a shared playlist

Input: @poster_ID, @post_date, @my_comment_text

Process: Users may comment in a particular post by pressing the 'Comment' button as shown in the picture above. After writing the comment the users must hit the ENTER key for the comment to be posted.

SQL Statements:
//insert comment into the comment_on table
INSERT INTO comment
VALUES (@poster_ID, @post_date, NOW(), @my_comment_text, @logged_in_userID, -1, NOW()
)

4.8.1 Making a reply on a comment

Input: @poster_ID, @post_date, @comment_date, @commenter_id, @my_comment_text

Process: Users may make a reply to a comment by clicking the 'Reply' button just below the respective comment. After they write the comment, they should press ENTER for the reply to be made

SQL Statements:
INSERT INTO comment
VALUES (@poster_ID, @post_date, @comment_date, @my_comment_text, @ commenter_id, @logged_in_userID, NOW() **)**

4.8.2 Making a like on a comment

Input: @posterID, @post_date, @commenter_id, @comment_date)

Process: Users may like a comment by pressing the yellow Like button on the right of the comment.

SQL Statements:
INSERT INTO comment_likes
VALUES(@logged_in_userID**,** @posterID, @post_date, @commenter_id, @comment_date, -1, NOW() **)**

4.8.3 Making a like on a post

Input: @postID, @post_date

Process: Users may like a post by pressing the Like button of the respective post.

SQL Statements:
INSERT INTO post_likes
VALUES(@logged_in_userID**,** @postID, @post_date**)**

4.9 Viewing a song

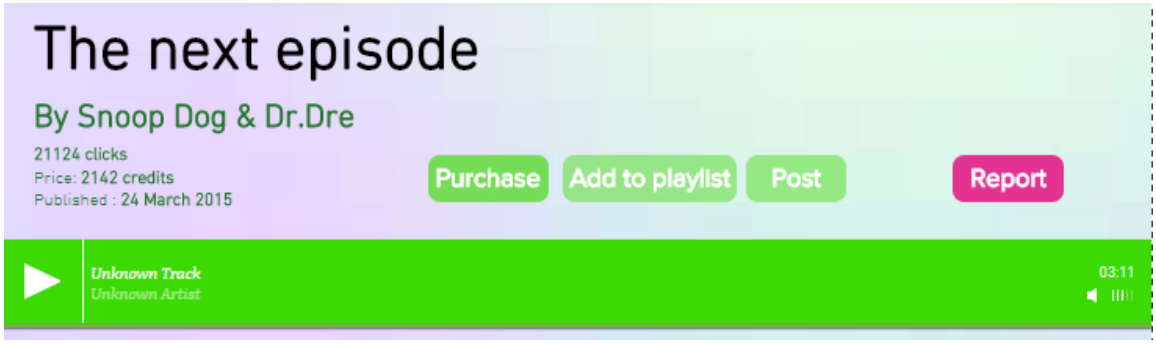

Figure 26- Song display

Input: @songID, @report_reason

Process: Users may choose to view songs by clicking on them in the search result, or in a post. The view is as shown above. Users may purchase the song, add it to a playlist, post it, or report it. When user clicks on report a small text box for writing the reason will show up.

Note: Purchase, Add to Playlist, Post are already defined queries.

SQL Statements:

// report a song
INSERT INTO report
VALUES (@logged_in_userID, @songID, @report_reason)

4.10 Rating a playlist

Figure 27-Playing and Rating playlists

Input: @playlistID, @rating

Process: Users may choose to rate a playlist by clicking on the RATE button of the respective playlist in the table shown in the picture above. Users cannot rate a playlist for more than once

SQL Statements:

// return all playlist id's already rated by the current user, so that user cannot rate them again
// user_rates_playlist is defined in section 5.12
SELECT *
FROM user_rates_playlist

INSERT INTO rate
VALUES (@logged_in_userID, @playlistID, @rating)

5.ADVANCED DATABASE COMPONENTS

Througout the database views and functions are used to make some sql statements easier to call and also easier for the user to comprehend. The @logged_in_user_ID is an input that represents the ID of the logged in user.

5.1 - This function return the number of users who have the same email or username. This is done to check for the uniqueness of the email and the username. Thus if this function returns 0 with the given parameters it means that there is no user with the same email and username.

```
create function check_user_exists(email varchar(40), username
varchar(40))
      returns integer
      begin
      declare amount integer;
            select count(*) into amount
            from User as U
            where U.email = email or U.username = username
      return amount;
      end
```

5.2 - This function returns the number of artists who have the same artist name or the same email. This is checked in order to verify that the artist has a unique artist name and a unique email.

```
create function check_artist_exists(art_name varchar(40), email
varchar(40))
      returns integer
      begin
      declare amount integer;
            select count(*) into amount
            from (User natural join Artist) as UA
            where UA.email = email or UA.art_name = art_name
      return amount;
```

```
end
```

5.3 - This function checks if the given email exists anywhere else except the given ID. If it does exist it means that it is already taken. This function is used in update to check when a particular user changes their email, if the email is used.

```
create function check_user_email(ID  integer, email varchar(40))
      returns integer
      begin
      declare amount integer;
            select count(*) into amount
            from User as U
            where U.email = email or U.ID <> ID
      return amount;
      end
```

5.4 - This function checks if the given artist nam and the email of that artist exist anywhere else except the given ID. This check is used when the artist updates the email or the artist name, in order to check if any of them is taken already.

```
create function check_artist_exists_update (ID integer, art_name
varchar(40),email varchar(40))
      returns integer
      begin
      declare amount integer;
            select count(*) into amount
            from (User natural join Artist) as UA
            where UA.ID <> ID and (UA.email = email or UA.art_name =
art_name)
      return amount;
      end
```

5.5 - Similarly to function at 5.4 this function checks if the production manager updating the label or email is actually trying to use one that is already in use. Thus the label given and the email given are checked if any of them belongs to another production manager with a different ID then the one given.

```
create function check_manager_exists_update (ID integer, label
varchar(40), email varchar(40))
	returns integer
	begin
	declare amount integer;
		select count(*) into amount
		from (User natural join Production_manager) as UA
		where UA.ID <> ID and (UA.email = email or UA.label_name
= label)
	return amount;
	end
```

5.6 - This view provides all the songs that the logged in user has purchased.

```
create view user_songs as
	select *
	from songs natural join purchase
	where purchase.ID = @logged_in_user_ID;
```

5.7 - This view provides all the artist IDs that the logged in user is following

```
create view user_follows_artist as
	select artist_ID
	from follows_artist
	where consumer_ID = @logged_in_user_ID;
```

5.8 - This view returns all the IDs of the albums that are purchased by a user. This is computed by using function 5.13 to get the total number of songs of a particular album and function 5.14 which returns the number of songs from an album. If these two are equal it means that the user has purchased the album as he owns all the songs.

```
create view user_albums as
	select ID
	from album
	where  get_num_songs_album(ID) =
get_num_album_user_specific(ID, @logged_in_user_ID);
```

5.9 - This view provides all the playlist IDs that the logged in user follows.

```
create view user_followed_playlists as
	select playlist_ID
	from follows_playlist
	where ID = @logged_in_user_ID;
```

5.10 - This view provides all the IDs of the consumers followed by the particular logged in consumer.

```
create view consumer_followed_consumer as
	select followed_consumer_ID
	from follows
	where following_consumer_ID = @logged_in_user_ID;
```

5.11 - This view provides the IDs of all the production managers that are followed by the logged in consumer.

```
create view user_follows_manager as
	select ID
	from follows_production_manager
	where consumer_ID = @logged_in_user_ID;
```

5.12 - This view provides all the playlist IDs rated by the logged in user.

```
create view user_rates_playlist as
      select playlist_ID
      from rate
      where ID = @logged_in_user_ID;
```

5.13 - This function returns the number of songs for the given album, which is passed as the album ID.

```
create function get_num_songs_album(album_ID integer)
      returns integer
      begin
      declare amount integer;
            select count(*) into amount
            from album_contains
            where album.album_ID = album_ID
      return amount;
      end
```

5.14 - This function returns the number of songs the given user owns from the given album. Both values are passed as their respective IDs.

```
create function get_num_album_user_specific(album_ID integer,
user_ID integer)
      returns integer
      begin
      declare amount integer;
            select count(*) into amount
            from album_contains natural join purchase
            where album.album_ID = album_ID and purchase.ID = user_ID
```

```
return amount;
end
```

5.15 - This view provides the information of all the users that the logged in user is allowed to see. This checks if the user has a privacy of shared whereby everyone is allowed to view them or the logged in user follows the particular user such that they can view their content.

```
create view visible_user_content as
    select *
    from User as U
    where U.user_privacy = 'public' or U.ID in (select
                                    followed_consumer_ID
                                from Follows
                                where following_consumer_ID =
                                @logged_in_user_ID);
```

5.16 - This function returns the information about the most liked reply for the given comment. First it counts all the likes each reply has got. Furthermore it checks if the replier ID is not equal to -1, because in that way it can differentiate between a comment and a reply. Next it orders everything and limits the ordering to show only the most liked comment based on the count done previously. In the end this function returns a table with a single row of information about the most liked reply.

```
create function most_liked_reply(ID integer, post_date timestamp,
comment_date timestamp, commenter_ID integer)
returns table(
        ID integer,
        post_date timestamp,
        comment_date timestamp,
        text text,
        commenter_ID integer,
        replier_ID integer,
```

```
            reply_date timestamp)
return table(
            select *
            from Comment as A
            where(A.creator_ID, A.post_date, A.comment_date,
A.commenter_ID, A.replier_ID, A.reply_date)
            in
                  (select  AC.creator_ID, AC.post_date,
                        AC.comment_date, AC.commenter_ID,
                        AC.replier_ID, AC.reply_date
                  from (select  C.creator_ID, C.post_date,
                              C.comment_date, C.commenter_ID,
                              C.replier_ID, C.reply_date, count(*) as
                              amount
                              from Comment_likes as C
                              where C.creator_ID = ID and
                              C.post_date = post_date and
                              C.comment_date = comment_date and
                              C.commenter_ID = commenter_ID and
                              C.replier_ID <> -1
                              group by  C.creator_ID, C.post_date,
                                    C.comment_date,  C.commenter_ID,
                                    C.replier_ID, C.reply_date)
                              order by amount desc
                              limit 1);
```

5.17 - This function returns a table which will contain the three most recent comments for a given post. It will use order by to sort them and then limit the results of the table to only 3. This is used when the user scrolls through his feed and just looks at the comments they can only see the three most recent, if they want to observe more they have to press see more.

```
create funciton three_recent_comments(creator_ID integer,
commenter_ID integer, post_date timestamp, comment_date timestamp)
      returns table(
            ID integer,
```

```
        post_date timestamp,
        comment_date timestamp,
        text text,
        commenter_ID integer,
        replier_ID integer,
        reply_date timestamp)
    return table
        (select *
        from Comment as C
        where C.creator_ID = creator_ID and
        C.post_date = post_date and
        C.commenter_ID = commenter_ID and
        C.comment_date = comment_date
        order by comment_date desc
        limit 3);
```

5.18 - This function returns the amount of users that follow the given user.

```
create function amount_followers (ID integer)
    returns integer
    begin
    declare amount integer;
        select count(*) into amount
        from Follows
        where ID = followed_consumer_ID
    return amount;
    end
```

5.19 - This function returns the age of the given user. It takes the current timestamp and takes the difference with the users birth date. The result is given in days and as such the resulting number is divided by 365, which is approximately the number of days per one year. In the end the floor of the result is taken as we want the current age which is the amount of total years passed completely. The ready functions used in this function are from mySQL, as it also is what we will use.

```
create function calc_age (user_ID integer)
      returns integer
      begin
      declare age integer;
            select floor ( datediff (now()  , C.birth_date) / 365)
into age
            from Consumer as C
            where C.ID = ID
      return age;
      end
```

5.20 - This function calculates the number of albums for a given artist.

```
create function nr_of_albums (artist_ID integer)
      returns integer
      begin
      declare amount integer;
            select count(*) into amount
            from Album
            where creator_ID = artist_ID
      return amount;
      end
```

5.21 - This function returns the number of followers an artist has.

```
create function num_followers_artist (artist_ID integer)
      returns integer
      begin
      declare amount integer;
            select count(*) into amount
```

```
            from Follows_artist as F
            where F.artist_ID = artist_ID
      return amount;
      end
```

5.22 - This function provides the average rating for a given playlist based on the ratings done by the users.

```
create function calc_rating( playlist_ID integer)
      returns double
      begin
      declare result double;
            select avg(rating) into result
            from rate as R
            where R.playlist_ID = playlist_ID
      return result;
      end
```

5.23 - This function returns the number of songs an artist has.

```
create function get_num_songs(artist_ID integer)
      returns integer
      begin
      declare amount integer;
            select count(*) into amount
            from creates natural join album_contains
            where creates.artist_ID = artist_ID
      return amount;
      end
```

5.24 - This function returns the number of projects a production manager has participated in.

```
create function manager_num_projects(manager_ID integer)
      returns integer
      begin
      declare amount integer;
            select count(*) into amount
            from creates
            where creates.manager_ID = manager_ID
      return amount;
      end
```

5.25 - This function returns the number of followers a production manager has

```
create function manager_num_followers(manager_ID integer)
      returns integer
      begin
      declare amount integer;
            select count(*) into amount
            from follows_production_manager as F
            where F.ID = manager_ID
      return amount;
      end
```

Limitations

Users can not register into the system if they do not verify their account through the email verification link within 24 hours.

A consumer can only view other consumers name, username, publicly shared playlists and date when that user registered. In order to show more information they should follow that user or that user should have a privacy setting of public

When users choose to delete their account, all of their personal information, songs purchased, playlists created, friends and artists followed will be deleted from the database system. This however doesn't affect the budget of the artist.

Each comment cannot have more than 200 characters.

Song names, album names, and playlist names cannot be longer than 40 characters

6. IMPLEMENTATION

MySQL will be used to implement this database system and PHP and AJAX will be used for the backend implementation. JSON objects corresponding to the models will be made in back end. For the front end we anticipate to use Bootstrap, HTML, CSS for UI and Javascript and PHP for controllers. Helper libraries will be used to connect the back end of the system with the database.