Bilkent University

Department of Computer Engineering

CS353 Course Project

TunedIn

Project Design Report

Furkan Kazım Akkurt, Cemal Arda Kızılkaya, Khasmamad Shabanovi, Mehmet Bora Kurucu

Instructor: Özgür Ulusoy
Teaching Assistant(s): Arif Usta

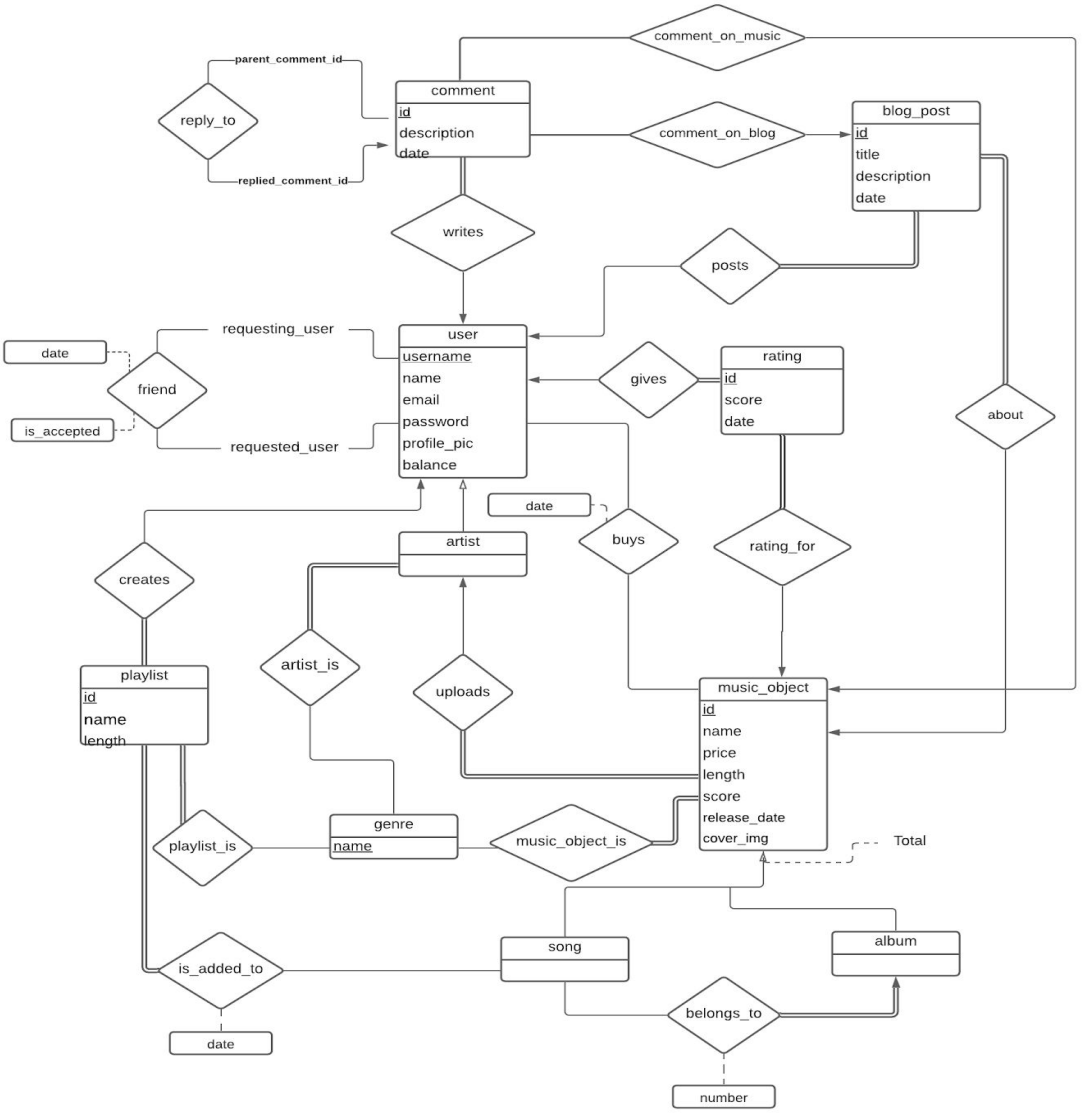
November 23, 2020

1. Revised E/R Model

Considering the feedback we got from the teaching assistant, we decided to add some new entities and update the existing ones for interactive functionality. Some of the modifications made are as follows:

- Changed the comment entity from weak entity to strong entity.
- Splitted the comment_on relation to two seperate relations named as comment_on_music and comment_on_blog, in order to prevent ambiguities regarding whether a comment is written to a music object or a blog post.
- In order to limit feedback with only score, changed the feedback entity a new entity called rating, which will only have the score and date of a rating.
- Changed the name of the relationship follows to friend.
- Changed the name of the relationship feedback_for to rating_for between rating and music_object.
- Added two new attributes to friend relation, named as date and is_accepted. Whether a friend request is accepted or not will be kept track of by is_accepted attribute.
- Added a new attribute called date to buys relation, specifying when a music_object is bought by a user.
- Added a new relation between blog_post and music_object called about. Within this relation, it can be seen what a specific blog post is written about.
- Removed the no_of_plays attribute of the song entity.
- Removed the bio attribute of the artist entity.
- Made the participation constraint of the playlist entity in is_added_to relationship total instead of partial.
- Made the participation constraint of the album entity in belongs_to relationship total instead of partial.
- Made the participation constraint of comment in comment_on_music relationship partial instead of total.

- Added a new relation called reply_to, in order to provide users functionality of being able to reply to a comment.
- Moved all attributes of album to its generalized entity, music_object.
- Deleted the redundant authors relation between music object and artist, since the same relation is implied by uploads as well.
- Added the number attribute to the belongs_to relation between song and album to keep track of the position of the song in the album.
- Added the date attribute to the is_added_to relationship between playlist and song to keep track of the order in which new songs are added into a playlist.



2. Relation Schemas

2.1. User

Relation Model:

User(<u>username</u>, name, email, password, profile_pic, balance)

Functional Dependencies:

username → name, email, password, profile_pic, balance
email → username, name, password, profile_pic, balance

Candidate Keys:

{(username), (name), (email)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE User(
        username        VARCHAR(255) PRIMARY KEY NOT NULL,
        name            VARCHAR(255) NOT NULL,
        email           VARCHAR(255) NOT NULL UNIQUE,
        password        VARCHAR(255) NOT NULL,
        profile_pic     VARCHAR(255),
        balance         DECIMAL(5,2) DEFAULT 0)
        Engine = InnoDB;
```

2.2. Artist

Relation Model:

Artist(artist_username)

Functional Dependencies:

None

Candidate Keys:

{(artist_username)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE Artist(
        artist_username         VARCHAR(255) PRIMARY KEY NOT NULL,
        FOREIGN KEY (artist_username) REFERENCES User(username))
        ENGINE = InnoDB;
```

2.3. Music Object

Relation Model:

MusicObject(music_object_id, name, price, length, score, release_date, cover_img, artist_username)

Functional Dependencies:

music_object_id → name, price, length, score, release_date, cover_img, artist_username
name, artist_username → music_object_id, price, length, score, release_date, cover_img

Candidate Keys:

{(music_object_id), (name, artist_username)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE MusicObject(
        music_object_id     INT PRIMARY KEY AUTO INCREMENT,
        name                VARCHAR(255) NOT NULL,
        price               DECIMAL(5,2) NOT NULL,
        length              INT,
        score               DECIMAL(3,1),
        release_date        DATE,
        cover_img           VARCHAR(255),
        artist_username     VARCHAR(255),
        FOREIGN KEY (artist_username) REFERENCES Artist(artist_username))
        ENGINE = InnoDB;
```

2.4. Album

Relation Model:

Album(album_id)

Functional Dependencies:

None

Candidate Keys:

{(album_id)}

Normal Form:

Boyce-Codd Normal Form

Table Definition:

```
CREATE TABLE Album(
        album_id        INT PRIMARY KEY,
        FOREIGN KEY (album_id) REFERENCES MusicObject(music_object_id))
        ENGINE = InnoDB;
```

2.5. Song

Relation Model:

Song(song_id, album_id, number)

Functional Dependencies:

song_id → album_id, number

Candidate Keys:

{(song_id)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE Song(
        song_id         INT PRIMARY KEY,
        album_id        INT,
        number          SMALLINT,
        FOREIGN KEY (song_id) REFERENCES MusicObject(music_object_id),
        FOREIGN KEY (album_id) REFERENCES Album(album_id))
        ENGINE = InnoDB;
```

2.6. Playlist

Relation Model:

Playlist(playlist_id, name, length, username)

Functional Dependencies:

playlist_id → length, name, username

Candidate Keys:

{(playlist_id)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE Playlist(
        playlist_id             INT PRIMARY KEY AUTO INCREMENT,
        name                    VARCHAR(255) NOT NULL,
        length                  INT DEFAULT 0,
        username                VARCHAR(255),
        FOREIGN KEY (username) REFERENCES User(username))
        ENGINE = InnoDB;
```

2.7. Rating

Relation Model:

Rating(<u>rating_id</u>, score, date, username, music_object_id)

Functional Dependencies:

rating_id → score, date, username, music_object_id
music_object_id, username → score, date, rating_id

Candidate Keys:

{(rating_id), (music_object_id, username)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE Rating(
        rating_id           INT PRIMARY KEY AUTO INCREMENT,
        score               TINYINT,
        date                DATE,
        username            VARCHAR(255),
        music_object_id     INT,
        FOREIGN KEY (username) REFERENCES User(username),
        FOREIGN KEY (music_object_id) REFERENCES MusicObject(music_object_id))
        ENGINE = InnoDB;
```

2.8. BlogPost

Relation Model:

BlogPost(blog_post_id, title, description, date, username, music_object_id)

Functional Dependencies:

blog_post_id → title, description, date, username, music_object_id
title, description, username → date, music_object_id

Candidate Keys:

{(blog_post_id), (title, description, username)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE BlogPost(
        blog_post_id        INT PRIMARY KEY AUTO INCREMENT,
        title               VARCHAR(255),
        description         TEXT,
        date                DATETIME,
        username            VARCHAR(255),
        music_object_id     INT,
        FOREIGN KEY (username) REFERENCES User(username),
        FOREIGN KEY (music_object_id) REFERENCE MusicObject(music_object_id))
        ENGINE = InnoDB;
```

2.9. Genre

Relation Model:

Genre(name)

Functional Dependencies:

None

Candidate Keys:

{(name)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE Genre(
        name   ENUM('Classical', 'Country', 'Electronic Dance', 'Hip-hop', 'Indie', 'Jazz',
        'K-pop', 'Metal', 'Oldies', 'Pop', 'Rap', 'Rythm & Blues', 'Rock', 'Techno') PRIMARY
        KEY
        ) ENGINE = InnoDB;
```

2.10. Comment

Relation Model:

Comment(comment_id, description, date, username, blog_post_id, music_object_id)

Functional Dependencies:

comment_id → description, date, username, blog_post_id, music_object_id
description, blog_post_id, username → comment_id, date
description, music_object_id, username → comment_id, date

Candidate Keys:

{(comment_id), (description, blog_post_id, username), (description, music_object_id, username)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE Comment(
        comment_id          INT PRIMARY KEY AUTO INCREMENT,
        description         TEXT,
        date                DATETIME,
        username            INT,
        blog_post_id        INT,
        music_object_id     INT,
        FOREIGN KEY (blog_post_id) REFERENCES BlogPost(blog_post_id),
        FOREIGN KEY (music_object_id) REFERENCES MusicObject(music_object_id)
        ) ENGINE = InnoDB;
```

2.11. ReplyTo

Relation Model:

ReplyTo(reply_comment_id, parent_comment_id)

Functional Dependencies:

None

Candidate Keys:

{(reply_comment_id)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE ReplyTo(
        reply_comment_id        INT PRIMARY KEY,
        parent_comment_id       INT,
        FOREIGN KEY (parent_comment_id) REFERENCES Comment(comment_id),
        FOREIGN KEY (reply_comment_id) REFERENCES Comment(comment_id))
        ENGINE = InnoDB;
```

2.12. Friend

Relation Model:

Friends(requesting_username, requested_username, request_date, is_accepted)

Functional Dependencies:

None

Candidate Keys:

{(requesting_username, requested_username)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE Friend(
        requesting_username  VARCHAR(255),
        requested_username   VARHCAR(255),
        request_date         DATETIME,
        is_accepted          TINYINT(1),
        PRIMARY KEY (requesting_username, requested_username),
        FOREIGN KEY (requesting_id) REFERENCES User(username),
        FOREIGN KEY (requested_id) REFERENCES User(username)
        ) ENGINE = InnoDB;
```

2.13. Music Object Genre

Relation Model:

MusicObjectGenre(music_object_id, genre_name)

Functional Dependencies:

None

Candidate Keys:

{(music_object_id, genre_name)}

Normal Form:

Boyce-Codd Normal Form

Table Definition:

```
CREATE TABLE MusicObjectGenre(
        music_object_id             INT,
        genre_name                  ENUM('Classical', 'Country', 'Electronic Dance',
        'Hip-hop', 'Indie', 'Jazz', 'K-pop', 'Metal', 'Oldies', 'Pop', 'Rap', 'Rythm & Blues', 'Rock',
        'Techno'),
        PRIMARY KEY (music_object_id, genre_name),
        FOREIGN KEY (music_object_id) REFERENCES MusicObject(music_object_id),
        FOREIGN KEY (genre_name) REFERENCES Genre(genre_name)
        ) ENGINE = InnoDB;
```

2.14. Artist Genre

Relation Model:

ArtistGenre(artist_username, genre_name)

Functional Dependencies:

None

Candidate Keys:

{(artist_username, genre_name)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE ArtistGenre(
        artist_username         VARCHAR(255),
        genre_name              ENUM('Classical', 'Country', 'Electronic Dance',
        'Hip-hop', 'Indie', 'Jazz', 'K-pop', 'Metal', 'Oldies', 'Pop', 'Rap', 'Rythm & Blues', 'Rock',
        'Techno'),
        PRIMARY KEY (artist_username, genre_name),
        FOREIGN KEY (artist_username) REFERENCES Artist(artist_username),
        FOREIGN KEY (genre_name) REFERENCES Genre(genre_name)
        ) ENGINE = InnoDB;
```

2.15. Buys

Relation Model:

Buys(<u>username, music_object_id,</u> date)

Functional Dependencies:

username, music_object_id → date

Candidate Keys:

{(username, music_object_id)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE Buys(
        username            VARCHAR(255),
        music_object_id     INT,
        date                DATETIME NOT NULL,
        PRIMARY KEY (username, music_object_id),
        FOREIGN KEY (username) REFERENCES User(username),
        FOREIGN KEY (music_object_id) REFERENCES MusicObject(music_object_id)
        ) ENGINE = InnoDB;
```

2.16. Playlist Song

Relation Model:

PlaylistSongs(playlist_id, song_id, date)

Functional Dependencies:

playlist_id, song_id → date

Candidate Keys:

{(playlist_id, song_id)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

```
CREATE TABLE PlaylistSong(
        playlist_id     INT,
        song_id         INT,
        date            DATETIME,
        PRIMARY KEY (playlist_id, song_id),
        FOREIGN KEY (playlist_id) REFERENCES Playlist(playlist_id),
        FOREIGN KEY (song_id) REFERENCES Song(song_id)
        ) ENGINE = InnoDB;
```

2.17. Playlist Genre

Relation Model:

PlaylistGenre(playlist_id, genre_name)

Functional Dependencies:

None

Candidate Keys:

{(playlist_id, genre_name)}

Normal Form:

Boyce-Codd Normal Form (BCNF)

Table Definition:

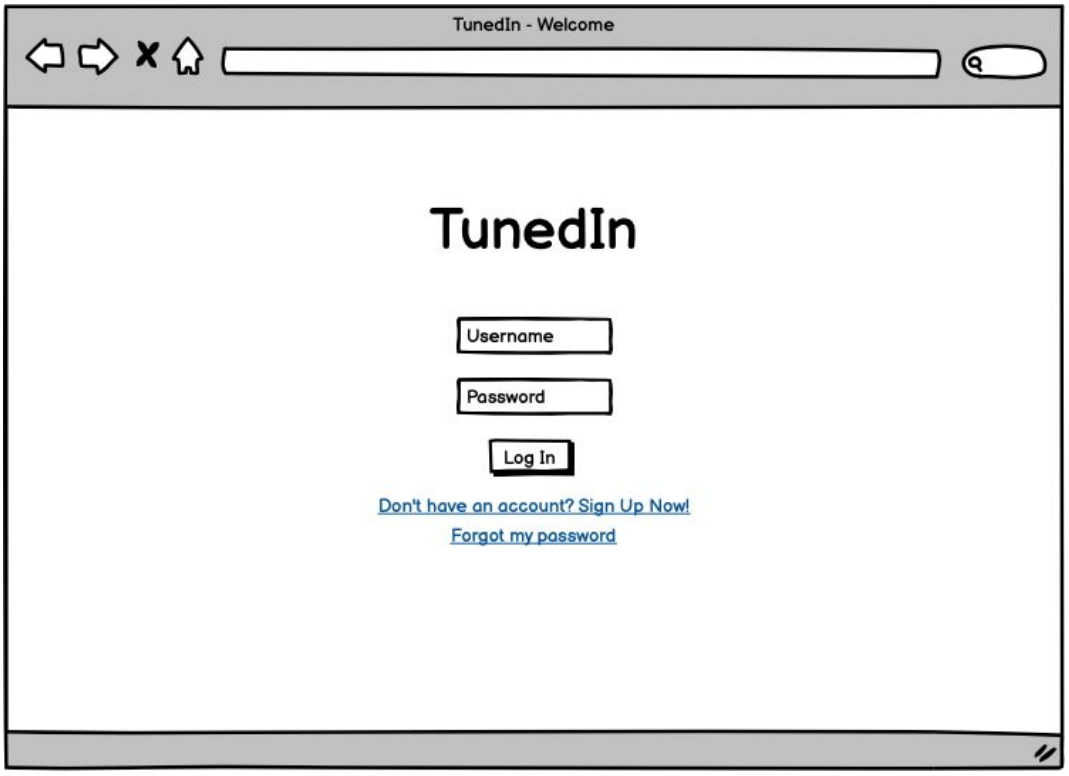
```
CREATE TABLE PlaylistGenre(
        playlist_id       INT,
        genre_name    ENUM('Classical', 'Country', 'Electronic Dance', 'Hip-hop', 'Indie',
        'Jazz', 'K-pop', 'Metal', 'Oldies', 'Pop', 'Rap', 'Rhythm & Blues', 'Rock', 'Techno'),
        PRIMARY KEY (playlist_id, genre_name),
        FOREIGN KEY (playlist_id) REFERENCES Playlist(playlist_id),
        FOREIGN KEY (genre_name) REFERENCES Genre(genre_name)
        ) ENGINE = InnoDB;
```

3. Functional Dependencies and Normalization of Tables

The previous section of the report contains all the information regarding the functional dependencies and normal forms for each individual table in the database. All relations are in Boyce-Codd Normal Form (BCNF). Since it is known that Third Normal Form (3NF) is the superset of BCNF, it can be said that all relations are in 3NF as well. Having said that, there was no need for any decomposition or normalization considering the fact that all relations were in BCNF.

4. User Interface Design and Corresponding SQL Statements

4.1. Log in



Inputs: @Username, @Password

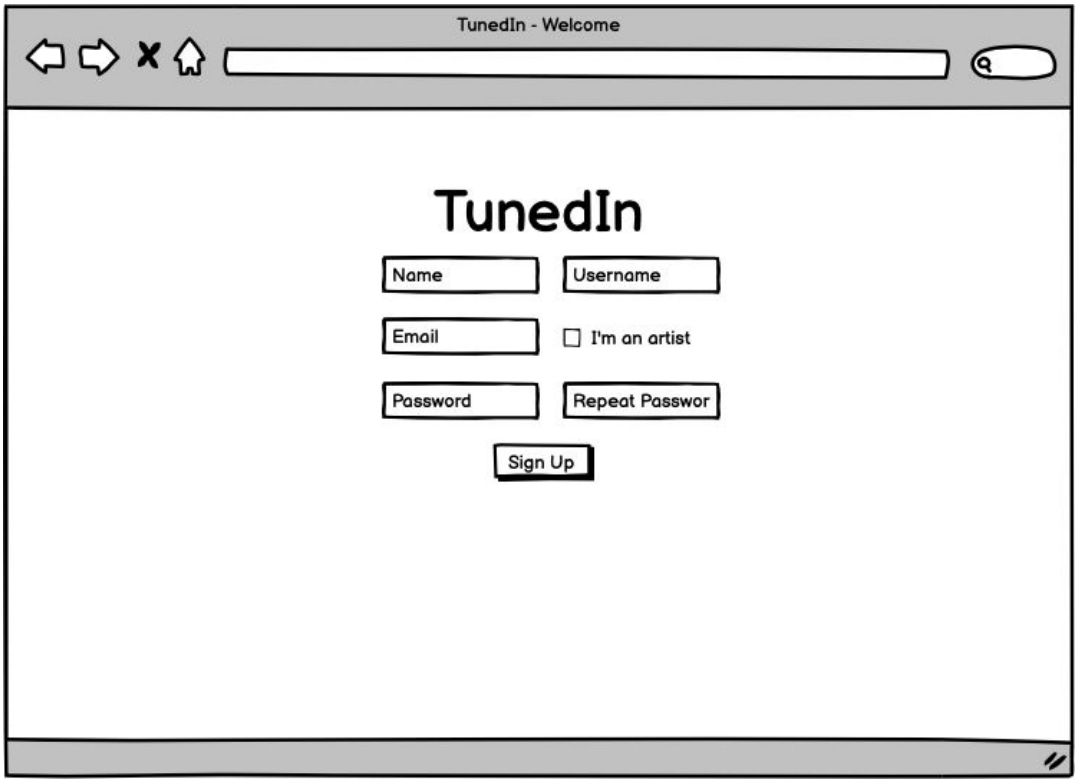
Process: Every registered user and newcomer first encounter this page. Current users of the system can register using their unique usernames and passwords, whereas newcomers can create an account by clicking the Sign Up link.

SQL Queries:

1) Log In

```
SELECT *
FROM User
WHERE username=@Username AND password=@Password;
```

4.2. Sign Up



Inputs: @Name, @Username, @Email, @Password, @RepeatPassword, @IsArtist

Process: Every newcomer, i.e. both standard users and artists, can create an account for free from the sign up page. If the email they specify doesn't conflict with any other email registered in the system and their username is unique, and the two password inputs are the same, the user will be successfully registered to the system, and their information will be recorded to the User table. Also, if the user is an artist, their username will be inserted into the Artist table to make our work regarding artists easier. If the above criteria aren't met, the registration will fail.

SQL Queries:

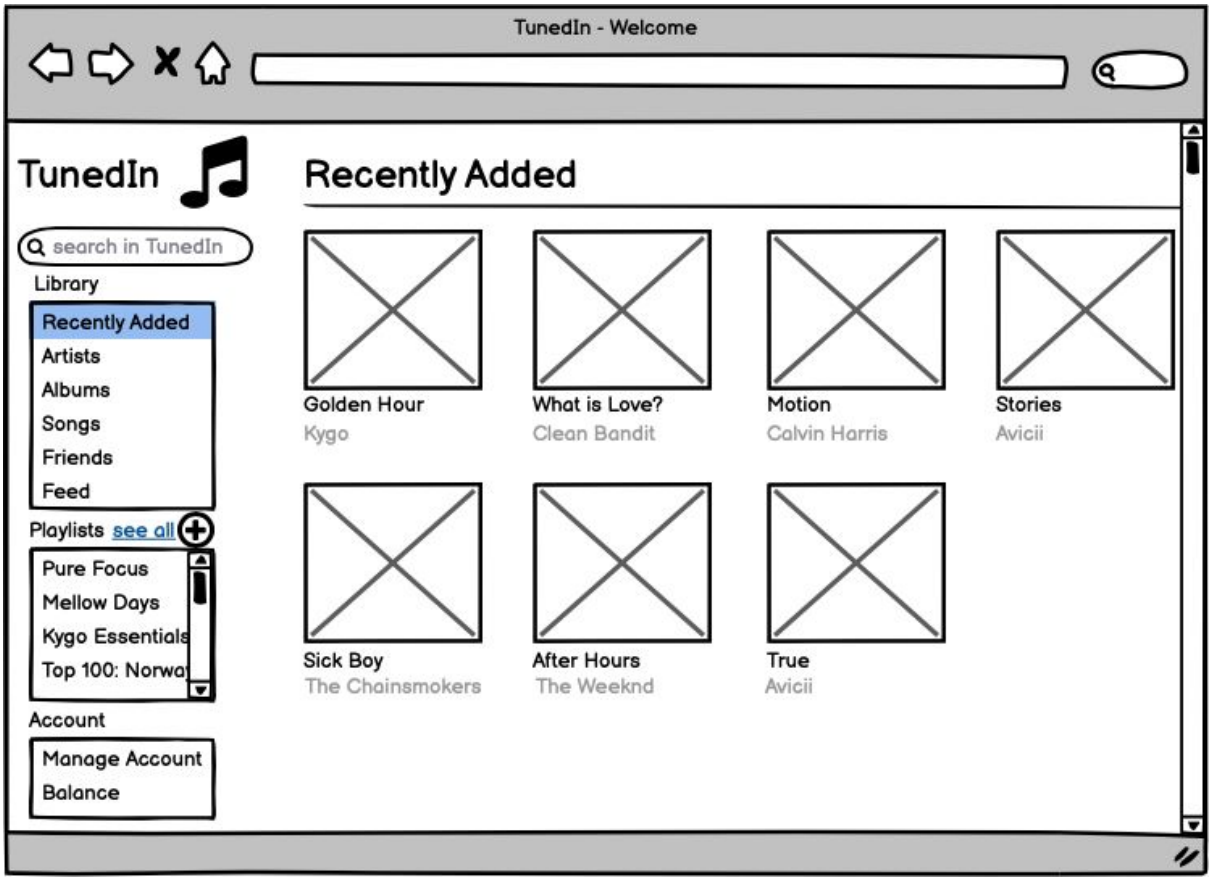
1) Sign Up for both standard users and artists

```
INSERT INTO User(username, name, email, password)
VALUES(@Username, @Name, @Email, @Password)
WHERE @Username NOT IN (Select username FROM User)
AND @Email NOT IN (SELECT email FROM User)
AND @Password=@RepeatPassword;
```

2) Additionally, add the username of artist to Artist table

```
SELECT
        CASE
                WHEN @IsArtist=1 THEN
                (INSERT INTO Artist(artist_username) VALUES(@Username))
        END;
```

4.3. Standard user home page/Recently added



Inputs: None

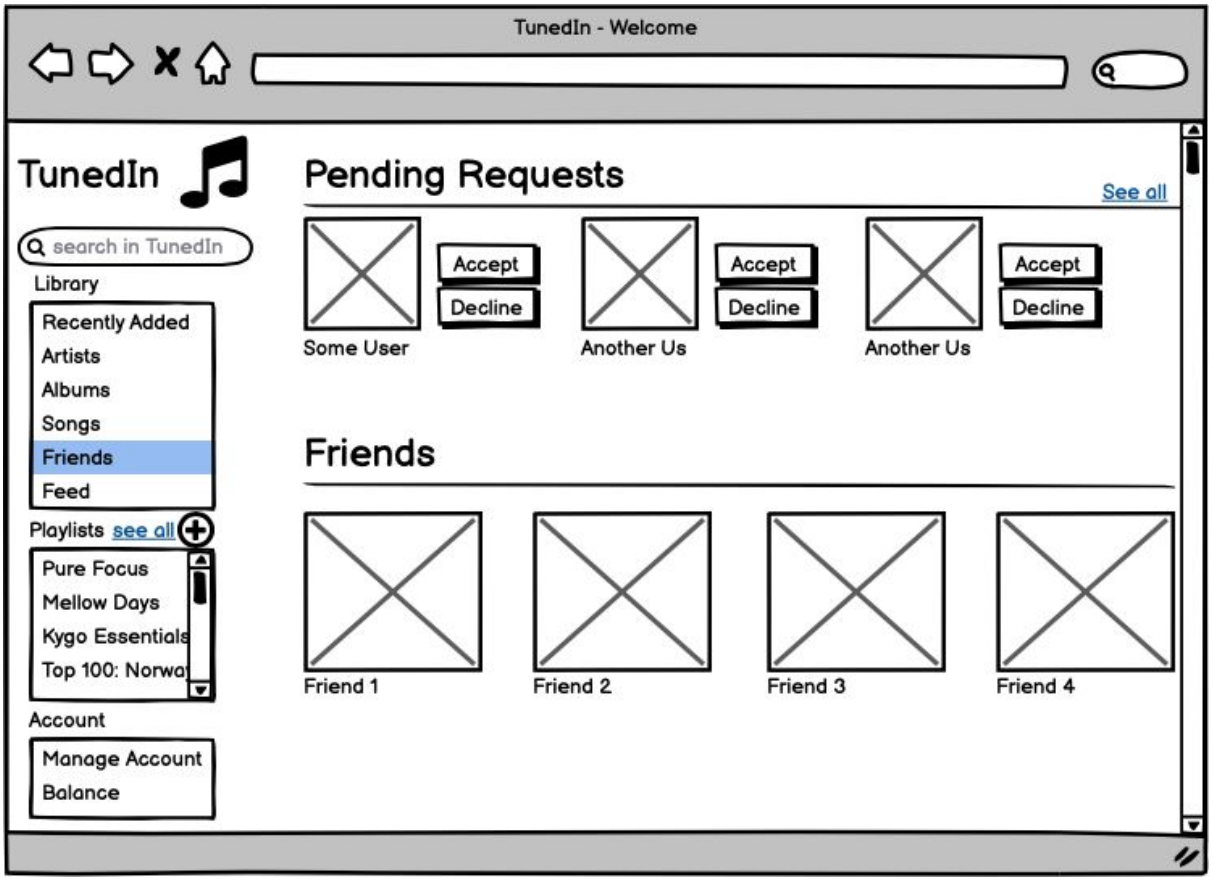
Process: Once logged in successfully, every user will see a screen where they can view at most 10 of their most recently purchased music files.

SQL Queries:

1) Retrieve most recent purchases of user

```
WITH AllPurchases(username, product_name, date) AS
        (SELECT artist_username, name, date
        FROM Buys NATURAL JOIN MusicObject
        WHERE username=@CurrentUsername)
        );

SELECT name, product_name
FROM AllPurchases NATURAL JOIN User
ORDER BY date DESC
LIMIT 10;
```

4.4. Friends



Inputs: @IsAccepted

Process: Once the users click the “Friends” tab in the tab bar, they can see their current friends and their waiting friend requests. They can simply accept or decline these requests by clicking the respective buttons.

SQL Queries:

1) Displaying the current friends of the user

```
SELECT name, username
FROM User
WHERE username IN
((SELECT requesting_username AS username
FROM friends
WHERE requested_username=@CurrentUsername AND is_accepted=1)
UNION
(SELECT requested_username AS username
FROM friends
WHERE requesting_username=@CurrentUsername AND is_accepted=1))
ORDER BY name ASC;
```

2) Displaying friend requests sent to them

```
SELECT User.username, name, date
FROM
(SELECT *
FROM User JOIN Friend ON User.username=Friend.requesting_username)
WHERE requested_username=@CurrentUsername
AND is_accepted=0
ORDER BY date DESC;
```

3) Accept or decline a friend request

```
UPDATE Friend
SET is_accepted=@IsAccepted
WHERE requested_username=@CurrentUsername
AND requesting_username=@SelectedUsername;
```

4.5. Songs

Inputs: None

Process: When the users click on the “Songs” button in the sidebar, they will be directed to a page where they can see all the songs they have purchased. The users can perform basic sorting on the songs they have purchased based on song name, artist, album name, and genre.

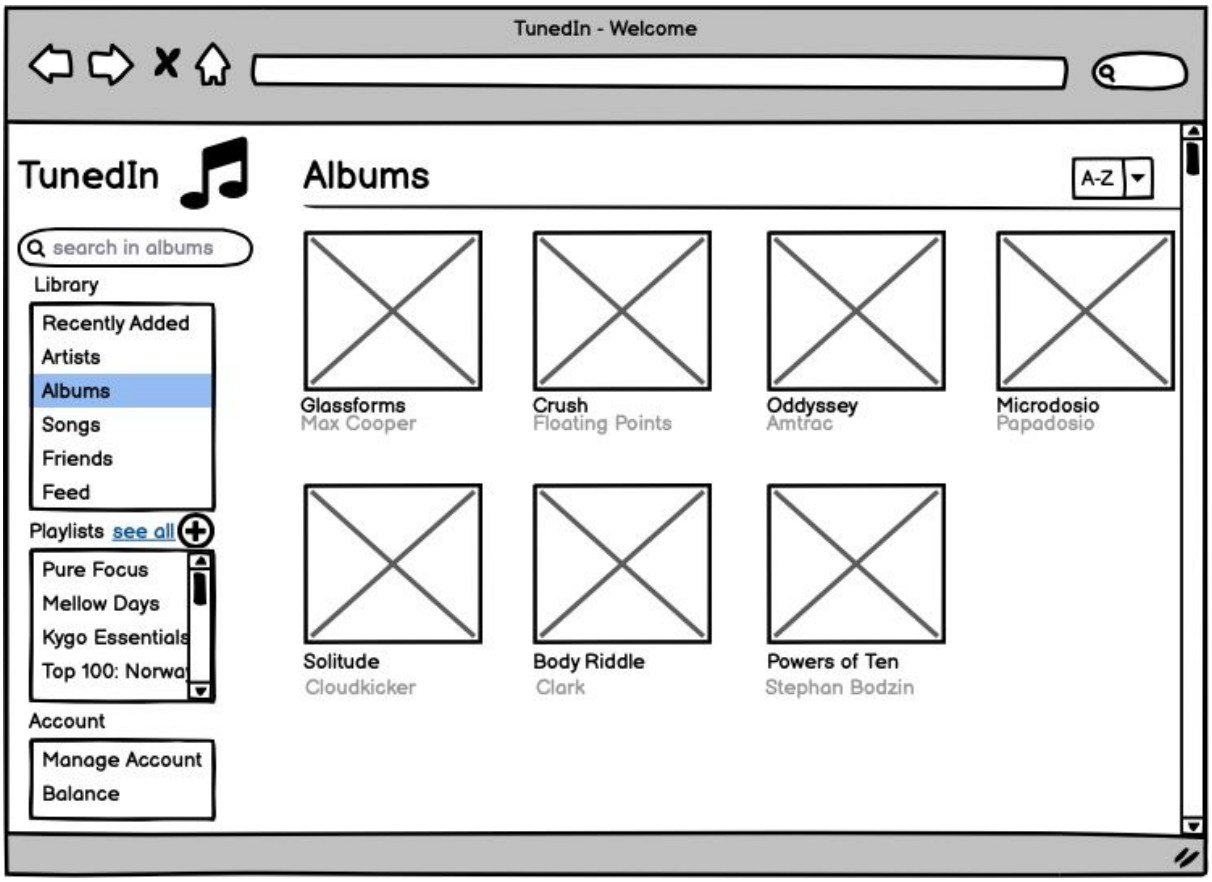
SQL Queries:

1) Displaying all the songs purchased by the user (either individually from an album or a single)

```
WITH Purchased(music_object_id) AS
                (SELECT music_object_id
                 FROM Buys
                WHERE username=@CurrentUsername);

WITH SingleIDs(music_object_id) AS
                (SELECT music_object_id
                 FROM Purchased
                 WHERE music_object_id IN (SELECT song_id FROM Song));
```

```
WITH Singles(name, username) AS
        (SELECT name, artist_username
        FROM MusicObject NATURAL JOIN SingleIDs);

WITH AlbumIDs(album_id) AS
        (Purchased EXCEPT SingleIDs);

WITH AlbumSongIDs(music_object_id) AS
        (SELECT song_id
        FROM AlbumIDs NATURAL JOIN Song);

WITH AlbumSongs(name, username) AS
        (SELECT name, artist_username
        FROM MusicObject NATURAL JOIN AlbumSongIDs);

SELECT name, User.name
FROM ((AlbumSongs UNION Singles)
    JOIN User USING(username))
ORDER BY name ASC;
```

4.6. Albums



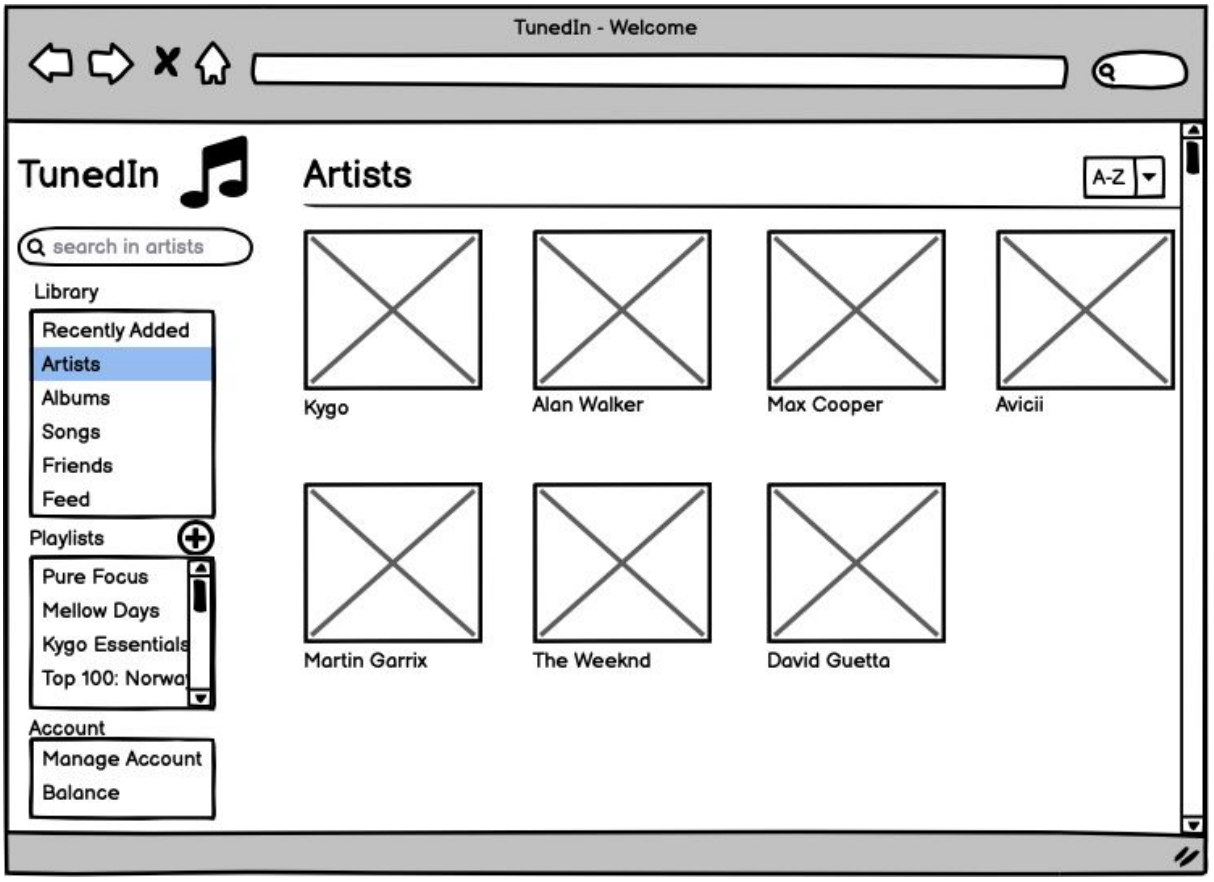
Inputs: None

Process: When the users click on the “Albums” button in the sidebar, they will be redirected to a page where they can see all albums they have purchased. They can sort the albums based on their names.

SQL Queries:

```
WITH PurchasedAlbums(music_object_id) AS
        ( SELECT music_object_id
          FROM Buys JOIN Album ON Buys.music_object_id=Album.album_id
          WHERE username=@CurrentUsername
        );

SELECT DISTINCT MusicObject.name, User.name, MusicObject.cover_img
FROM ((MusicObject NATURAL JOIN PurchasedAlbums) JOIN User
ON MusicObject.artist_username=User.username)
ORDER BY MusicObject.name ASC;
```

4.7. Artists



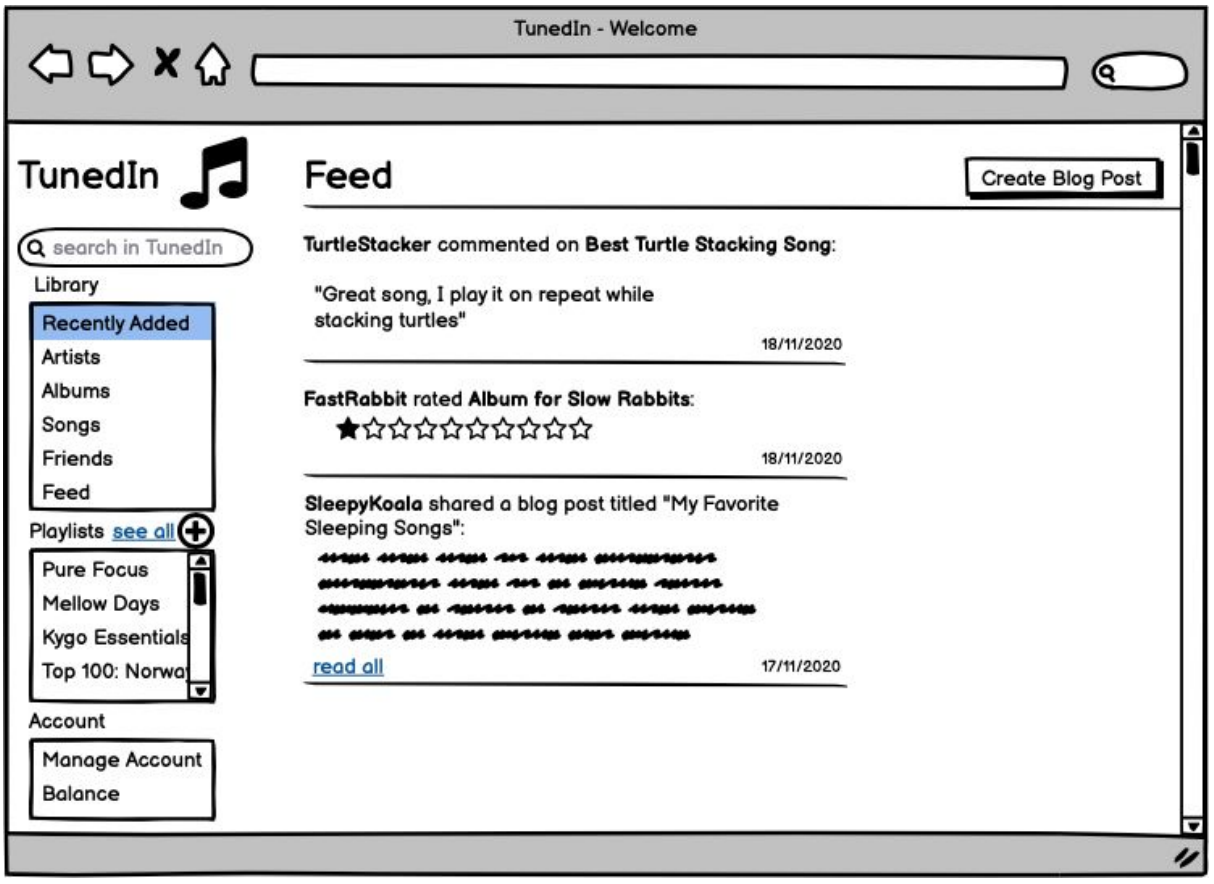
Inputs: None

Process: When the users click on the “Artists” button in the sidebar, they will be redirected to a page where they can see all the artists they have purchased any music file from. They can sort the artists based on their names.

SQL Queries:

```
SELECT DISTINCT name, username, profile_pic
FROM(
            (SELECT artist_username
            FROM Buys NATURAL JOIN MusicObject
            WHERE username=@CurrentUsername)
            JOIN User ON User.username=artist_username
    )
ORDER BY name ASC;
```

4.8. Feed



Inputs: None

Process: When the users click on the “Feed” tab on the sidebar, they will be presented with a union of their and their friends’ recent activities on TunedIn such as commenting on a music file or creating a blog post.

SQL Queries:

1) Getting blog posts for Feed

```
WITH People(username) AS
        ((SELECT requested_username AS username
        FROM Friend
        WHERE requesting_username=@CurrentUsername AND is_accepted=1)
        UNION
        (SELECT requesting_username AS username
        FROM Friend
        WHERE requested_username=@CurrentUsername AND is_accepted=1)
        UNION
        (SELECT @CurrentUsername AS username));
```

```
WITH SelectedBlogPosts(title, description, date, username, music_object_id) AS
        (SELECT title, description, date, username, music_object_id
         FROM BlogPost NATURAL JOIN People);

SELECT MusicObject.name, title, description, date, username
FROM SelectedBlogPosts NATURAL JOIN MusicObject
ORDER BY date DESC;
```

2) Getting ratings for Feed

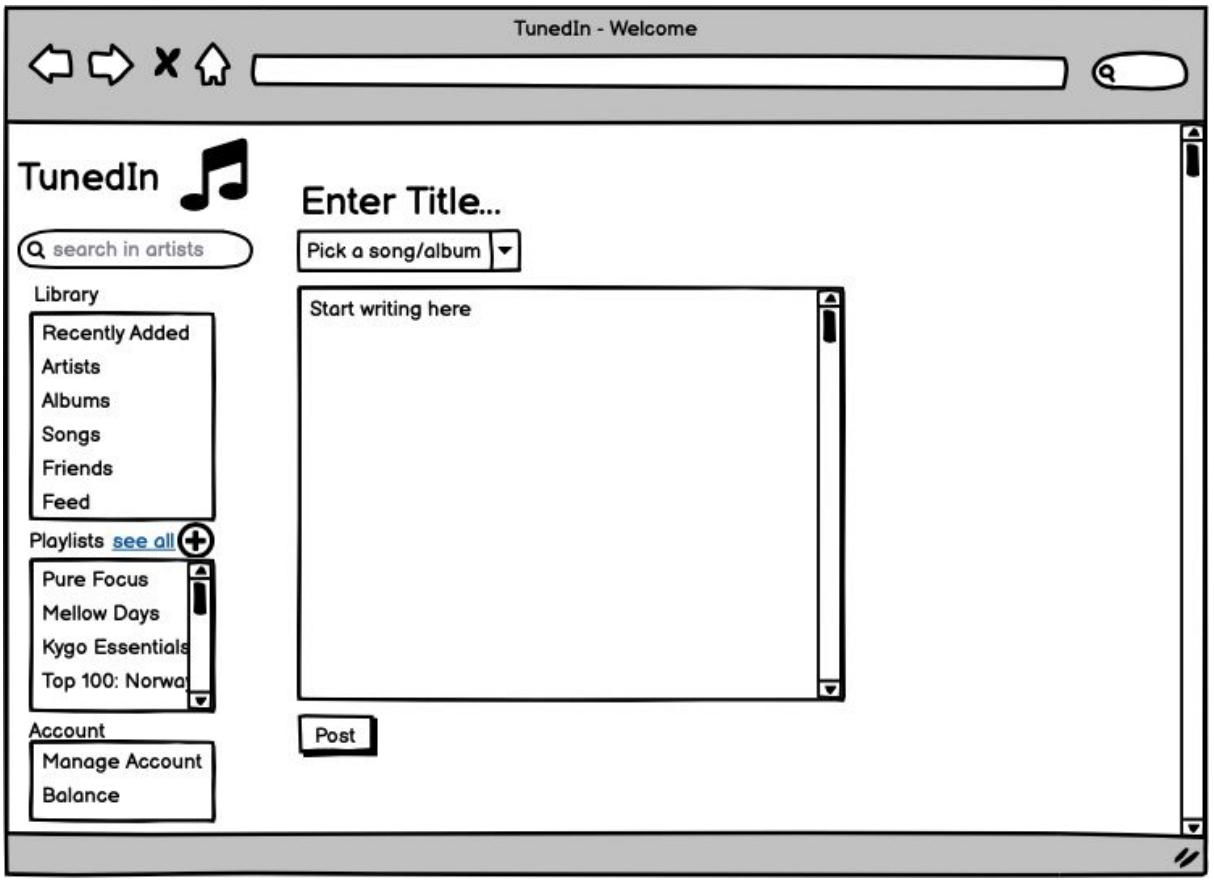
```
WITH People(username) AS
        ((SELECT requested_username AS username
        FROM Friend
        WHERE requesting_username=@CurrentUsername AND is_accepted=1)
        UNION
        (SELECT requesting_username AS username
        FROM Friend
        WHERE requested_username=@CurrentUsername AND is_accepted=1)
        UNION
        (SELECT @CurrentUsername AS username));

WITH SelectedRatings(score, date, username, music_object_id) AS
        (SELECT score, date, username, music_object_id
        FROM Rating NATURAL JOIN People);

SELECT MusicObject.name, score, date, username
FROM SelectedRatings NATURAL JOIN MusicObject
ORDER BY date DESC;
```

3) Getting comments for Feed

```
WITH People(username) AS
        ((SELECT requested_username AS username
        FROM Friend
        WHERE requesting_username=@CurrentUsername AND is_accepted=1)
        UNION
        (SELECT requesting_username AS username
        FROM Friend
        WHERE requested_username=@CurrentUsername AND is_accepted=1)
        UNION
        (SELECT @CurrentUsername AS username)
        );

WITH SelectedComments(description, date, username, blog_post_id,
music_object_id) AS
(SELECT description, date, username, blog_post_id, music_object_id
FROM Comment NATURAL JOIN Friends);
```

```
SELECT description, date, username, MusicObject.name
FROM SelectedComments NATURAL JOIN MusicObject
ORDER BY date DESC;

SELECT description, date, username, BlogPost.title
FROM SelectedComments NATURAL JOIN BlogPost
ORDER BY date DESC;
```

4.9. Create blog posts



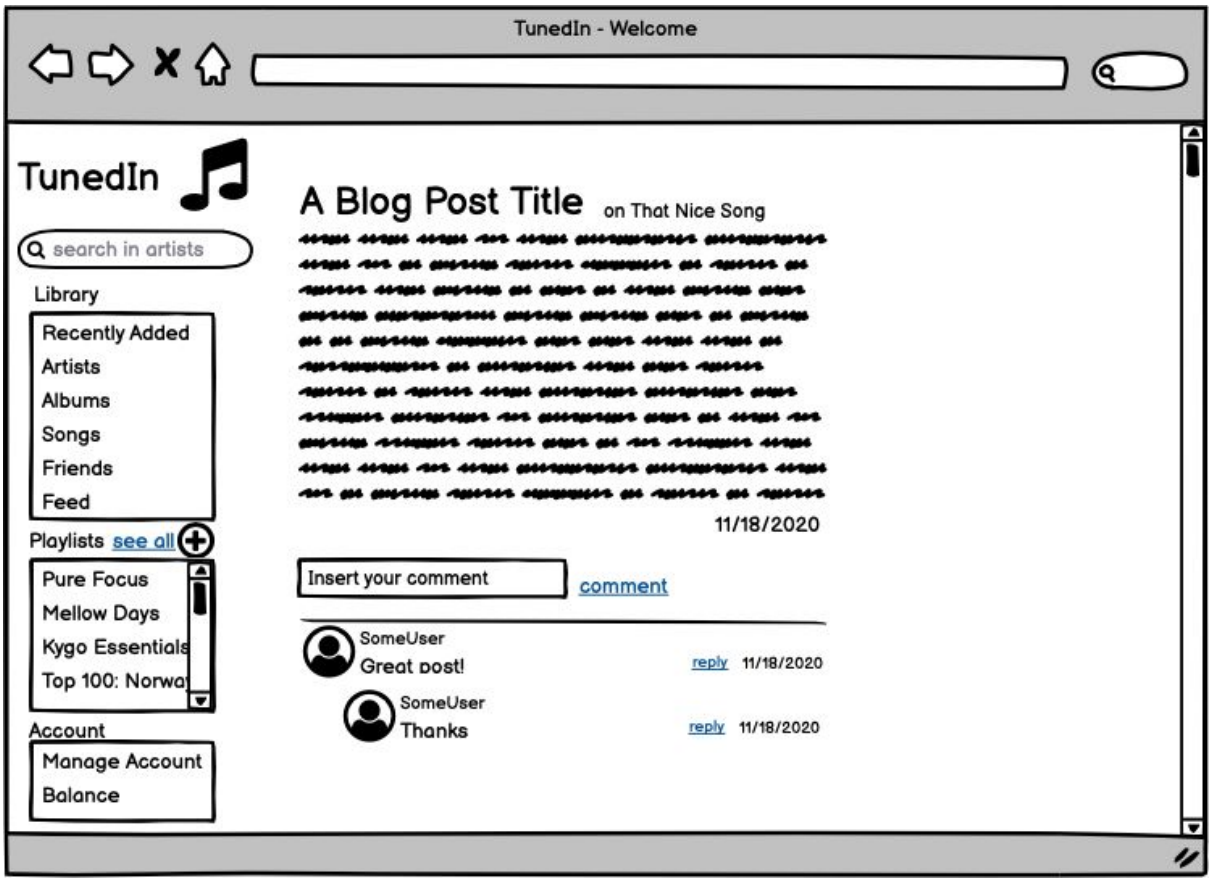
Inputs: @MusicObjectID, @BlogPostTitle, @BlogPostText

Process: By clicking the “Create Blog Post” button in their Feed, the users can write a blog post about a music file that they have already purchased.

SQL Queries:

INSERT INTO BlogPost(title, description, date, username, music_object_id)
VALUES(@BlogPostTitle, @BlogPostText, NOW(), @CurrentUsername, @MusicObjectID);

4.10. A blog post



Inputs: @Comment

Process: When the users select a blog post from their Feed, they can see the complete blog post and the comments on it. At this point, they can either make a comment directly by typing their comment into the text area just below the blog post, or they can reply to a comment by clicking “Reply”.

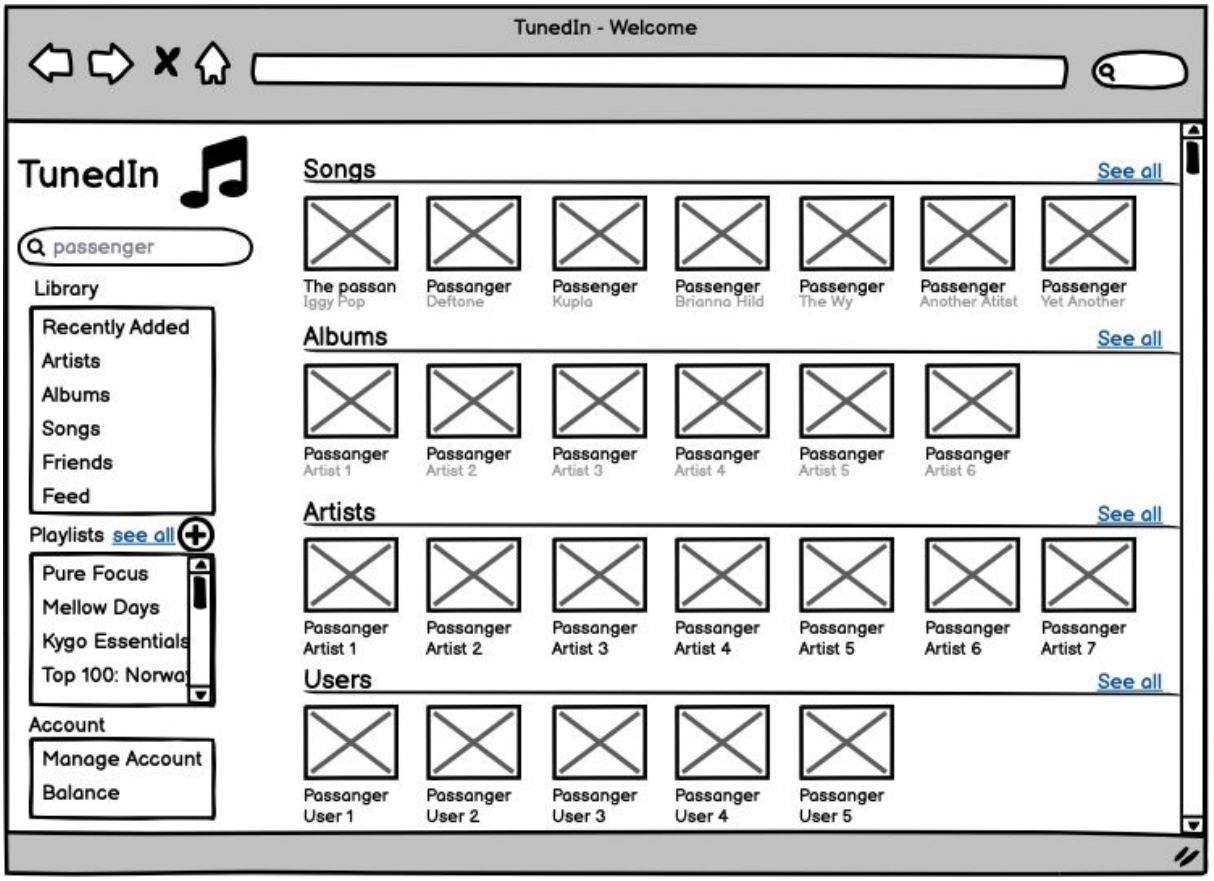
1) Commenting directly on a blog post

```
INSERT INTO
Comment(description, date, username, blog_post_id, music_object_id)
VALUES(@Comment, NOW(), @CurrentUsername, @CurBlogPostID,
@CurMusicObjectID);
```

2) Replying to a comment

```
INSERT INTO
Comment(description, date, username, blog_post_id, music_object_id)
VALUES(@Comment, NOW(), @CurrentUsername, @CurBlogPostID,
@CurMusicObjectID);

INSERT INTO ReplyTo(reply_comment_id, parent_comment_id)
VALUES(@ReplyID, @ParentCommentID);
```

4.11. Search results



Inputs: @SearchQuery

Process: When the users type in a keyword into the search bar and press enter, they will be presented with songs, albums, artists, and users that contain their search query.

SQL Queries:

1) Retrieving songs that contain the search query as a substring

```
WITH Songs(name, cover_img, username) AS
        (SELECT name, cover_img, artist_username
        FROM MusicObject NATURAL JOIN Song
        );

SELECT User.name, User.username, cover_img, Songs.name
FROM Songs  JOIN User USING(username)
WHERE Songs.name LIKE CONCAT('%', @SearchQuery, '%');
```

2) Retrieving albums that contain the search query as a substring

```
WITH Albums(name, cover_img, username) AS
        (SELECT name, cover_img, artist_username
        FROM MusicObject JOIN Album
        ON Album.album_id=MusicObject.music_object_id
        );

SELECT User.name, User.username, cover_img, Albums.name
FROM Albums  JOIN User USING(username)
WHERE Albums.name LIKE CONCAT('%', @SearchQuery, '%');
```

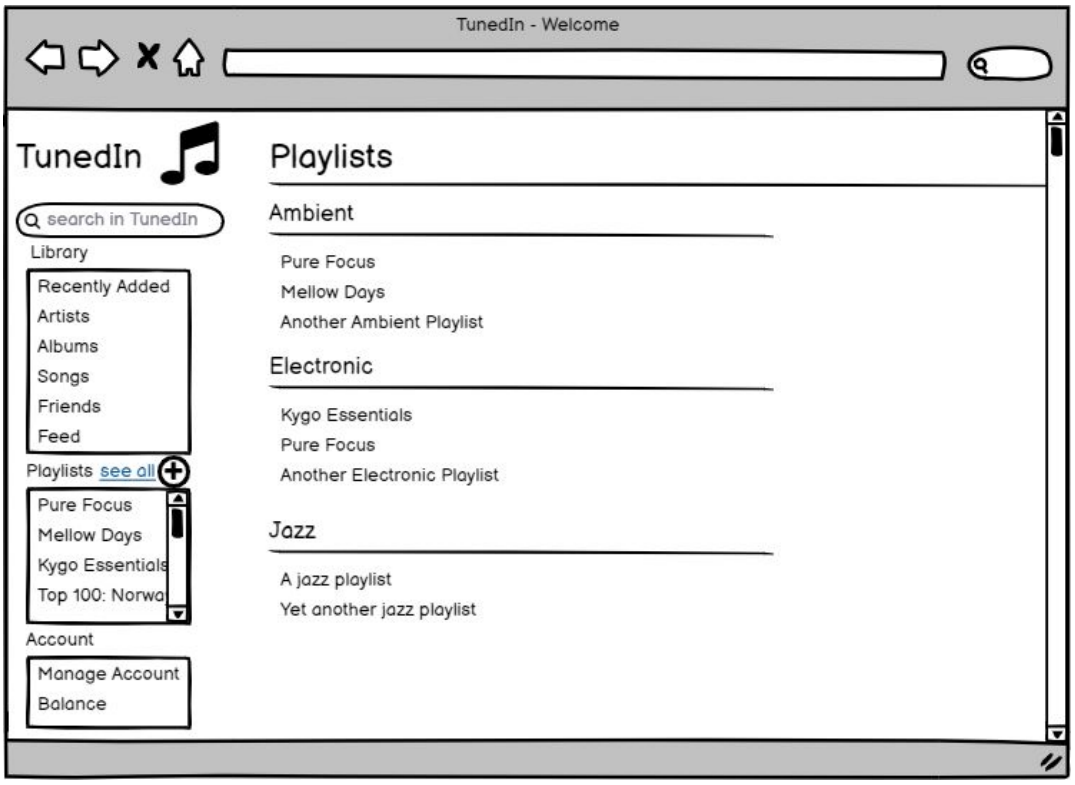
3) Retrieving artists whose names or usernames contain the search query as a substring

```
SELECT name, username, profile_pic
FROM User
WHERE username IN (SELECT artist_username FROM Artist) AND
(name LIKE CONCAT('%', @SearchQuery, '%') OR
username LIKE CONCAT('%', @SearchQuery, '%'));
```

4) Retrieving standard users whose names or usernames contain the search query as a substring

```
SELECT name, username, profile_pic
FROM User
WHERE username NOT IN (SELECT artist_username FROM Artist) AND
(name LIKE CONCAT('%', @SearchQuery, '%') OR
username LIKE CONCAT('%', @SearchQuery, '%'));
```

4.12. All playlists

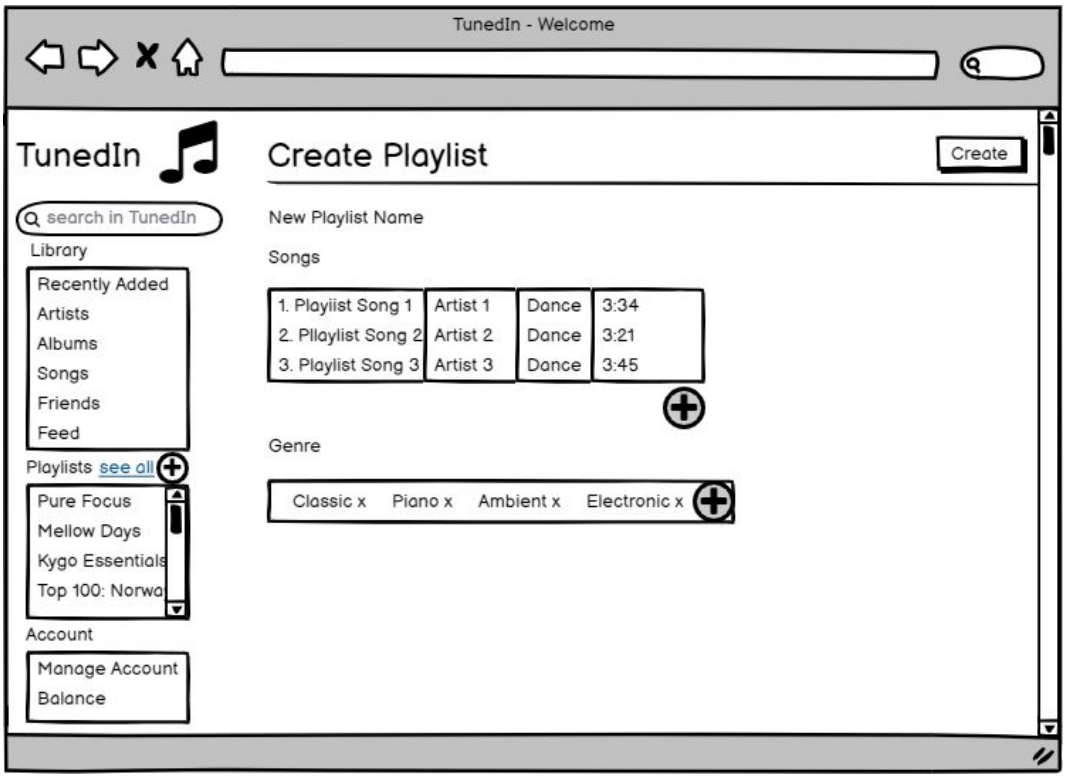


Process: Clicking on see all button in front of the Playlist on the left panel opens this page where the user can see all of her playlists categorized based on genres.

SQL Queries:

```
SELECT *
FROM playlist natural join playlistGenre
WHERE playlist.username = @username;
```

4.13. Create a playlist



Inputs: @playlist_name, @genre_name, @song_id

Process: Clicking on the '+' button in front of playlists in the left panel brings the user to this page where she can input details for creating a new playlist.

SQL Queries:

1) Show all the genres to user to choose from:

```
SELECT *
FROM genre;
```

2) Creating a new playlist:

```
INSERT INTO playlist (name, username)
values(@playlist_name, @username);
```

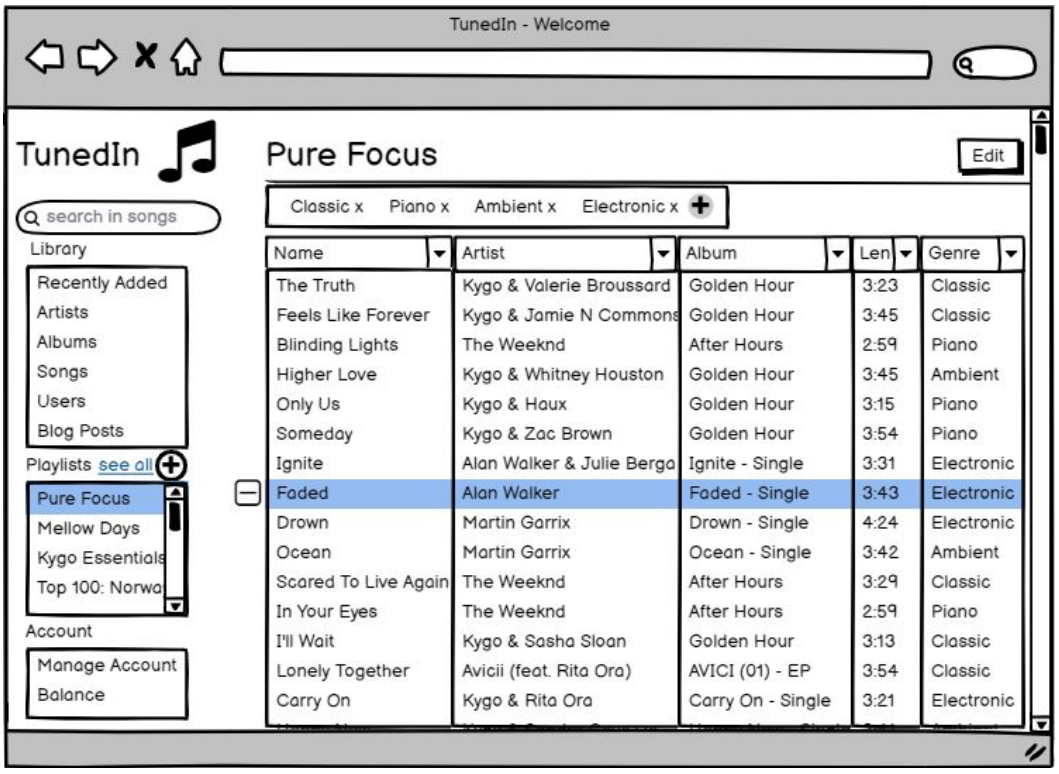
3) Adding a genre to the playlist (@playlist_id refers to the id of the newly generated playlist):

```
INSERT INTO playlistGenre
values (@playlist_id, @genre_name);
```

4) Adding a song to the playlist (@song_id refers to the id of the selected song):

```
INSERT INTO playlistSong
values(@playlist_id, @song_id, CURRENT_TIMESTAMP);
```

4.14. A playlist



Inputs: @playlist_id, @song_id

Process: Clicking on a specific playlist name redirects the user to this page where all the songs in the playlist are listed together with the genres of the playlist. Right clicking on a specific song reveals the '-' button which upon clicking removes the song from the playlist. Clicking on the arrow corresponding to the name of one of the columns sorts the songs according to the values of that column.

SQL Queries:

1) Getting the name of the selected playlist (referred with @playlist_id):

```
SELECT name
FROM playlist
WHERE playlist_id = @playlist_id;
```

2) Getting all the genres of the playlist:

```
SELECT *
FROM playlistGenre
WHERE playlist_id = @playlist_id;
```

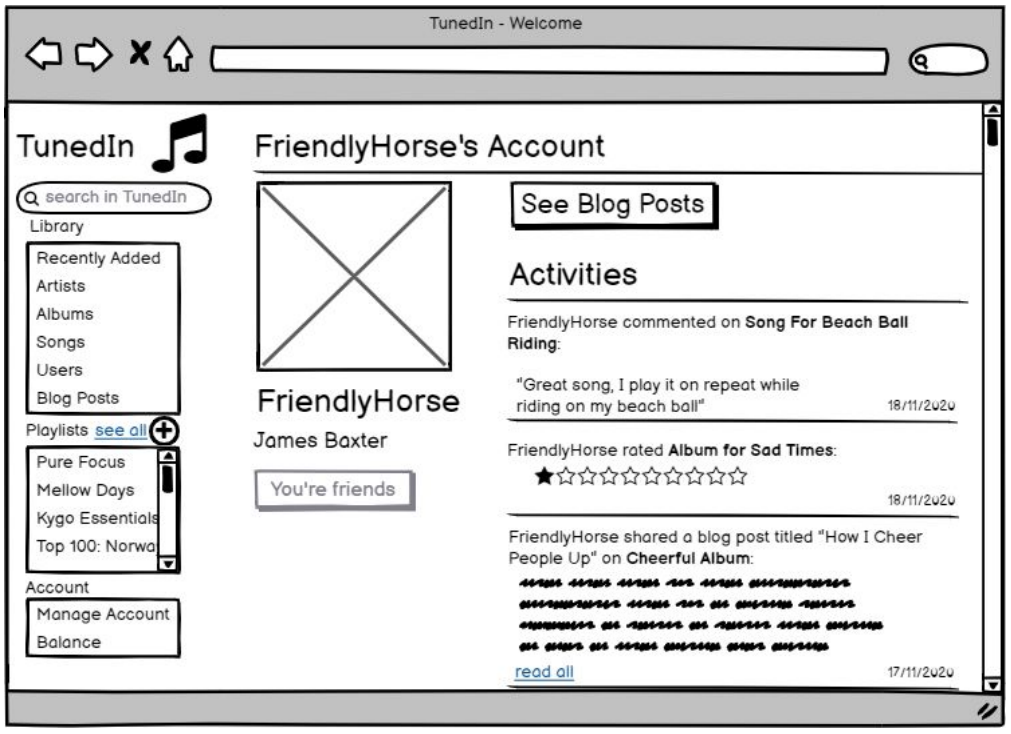
3) Getting all the songs in the playlist with album, genre, and artist information:

```
SELECT mo1.name as song_name,
mo2.name as album_name,
mo1.artist_username as artist_name,
mo1.length as len,
mog.genre_name as genre,
playlistSongs.date as date
FROM playlistSongs, song, musicObject mo1,
musicObject mo2, musicObjectGenre mog
WHERE playlistSongs.playlist_id = @playlist_id and
        song.song_id = playlistSongs.song_id and
        song.song_id = mo1.music_object_id and
        song.album_id = mo2.music_object_id and
        mog.music_object_id = song.song_id;
```

4) Removing the selected song (referred with @song_id) from the playlist:

```
DELETE FROM playlistSongs
WHERE song_id = @song_id;
```

4.15. Friend profile



Inputs: @friend_username

Process: This is how the profile of a friend is presented to the user. The user can view the username, name, and the profile picture of her friend in addition to the list of her friend's activities. The user can not see another user's activities unless they are friends. In case the users are not friends, then the disabled "You're friends" button appears as "Send request". Pressing the "Send request" button, the user can send a friendship request.

SQL Queries:

1) Getting basic user info:

```
SELECT name, username, profile_pic
FROM user
WHERE user.username = @friend_username;
```

2) Getting friend's comments on blog posts:

```
SELECT description, bp.description as bpdesc,
title, comment.date as date
FROM comment, blogpost bp
WHERE comment.username = @friend_username and
comment.blog_post_id = bp.blog_post_id
ORDER BY date;
```

3) Getting friend's comments on music objects:

```
SELECT description, mo.name, date
FROM comment, musicObject mo
WHERE comment.username = @friend_username and
comment.music_object_id = mo.music_object_id
ORDER BY date;
```

4) Getting friend's replies to comments on music objects:

```
SELECT description, mo.name, date
FROM replyTo, comment, musicObject mo
WHERE replyTo.reply_comment_id = comment.comment_id and
comment.username = @friend_username and
comment.music_object_id = mo.music_object_id
ORDER BY date;
```

5) Getting friend's replies to comments on Blog posts:

```
SELECT description, bp.description as bpdesc,
title, comment.date as date
FROM replyTo, comment, blogpost bp
WHERE replyTo.reply_comment_id = comment.comment_id and
comment.username = @friend_username and
comment.blog_post_id = bg.blog_post_id
ORDER BY date;
```

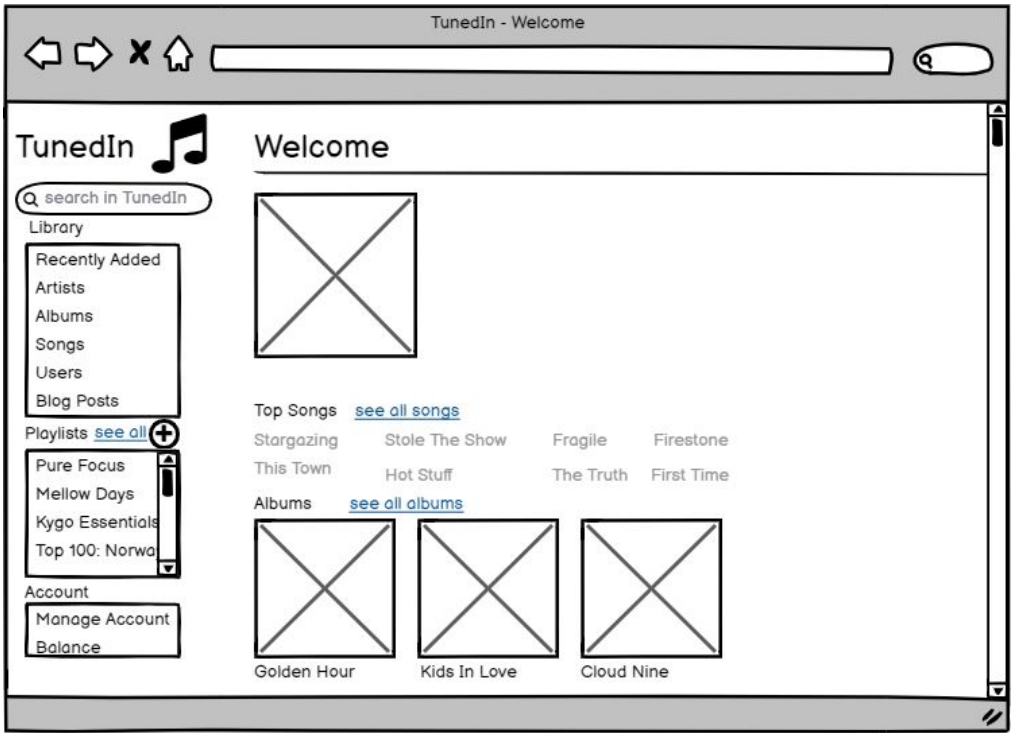
6) Getting friend's ratings:

```
SELECT mo.name as name, score, date
FROM rating, musicObjecet mo
WHERE rating.username = @friend_username and
rating.music_object_id = mo.music_object_id
ORDER BY date;
```

7) Sending a friendship request:

```
INSERT INTO friend
values(@username, @friend_username, CURRENT_TIMESTAMP, 0);
```

4.16. Artist homepage



Inputs: @artist_username

Process: This is the welcome/home page for the artist. The artist can view her top songs and albums together with her profile picture. Clicking on “see all songs” and “see all albums” buttons redirects the artist to a page where she sees a full list of her songs and albums respectively. This is also what another user sees when viewing an artist’s profile.

SQL Queries:

1) Getting the profile picture of the artist

```
SELECT profile_pic
FROM user
WHERE username = @artist_username;
```

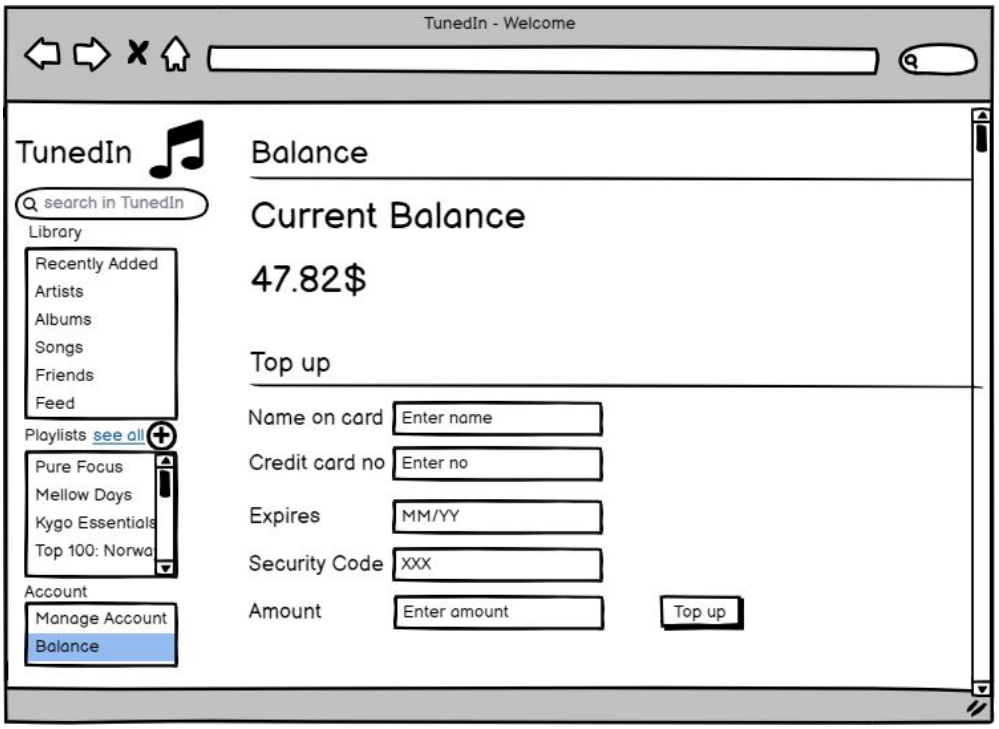
2) Getting the top songs

```
SELECT name, score
FROM song, musicObject mo
WHERE song.song_id = mo.music_object_id and
mo.artist_username = @artist_username
ORDER BY score DESC;
```

3) Getting the top albums

```
SELECT name, cover_img, score
FROM album, musicObject mo
WHERE album.album_id = mo.music_object_id and
mo.artist_username = @artist_username;
```

4.17. Balance



Inputs: @amount

Process: On this page, the user can see the current amount in her balance. Also, by entering her credit card details and the amount, the user can top up her balance.
SQL Queries:

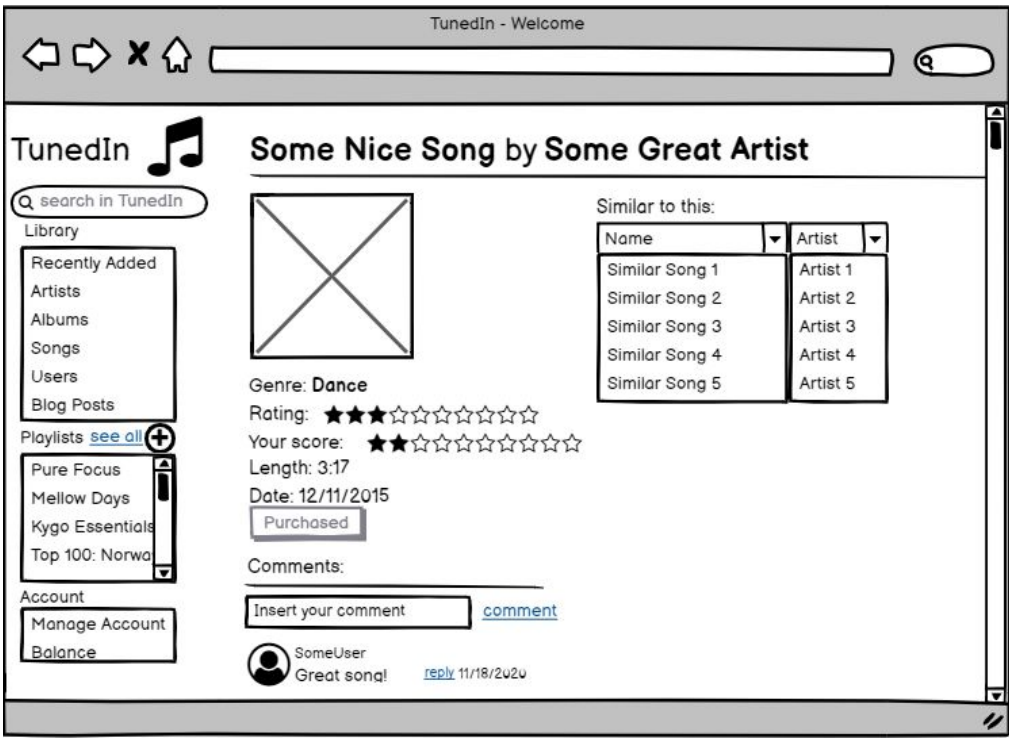
1) Getting the balance:

```
SELECT balance
FROM user
WHERE username = @username;
```

2) Updating the balance:

```
UPDATE user
SET balance = balance + @amount
WHERE username = @username;
```

4.18. Song profile



Inputs: @song_id, @comment_desc, @parent_comment_id

Process: On the song profile page, the user can view the name, artist, genre, length, overall rating, the date release, price, and the cover img of the song. Moreover, the user can see her rating of the song (if available) and the comments on the songs by other users. The user is also provided with a list of similar songs based on the genre ordered by the rating. If the user has purchased the song, she can add a comment or reply to other comments.

SQL Queries:

1) Getting basic song info:

```
SELECT mo.name as songName, mo.artistusername as artistName,
mo.genre as genre, length, release_date, score, cover_img,
price, (SELECT CASE WHEN EXISTS(
                                        SELECT *
                                        FROM buys
                                        WHERE username = @username and
                                                music_object_id = @song_id
                                ) THEN 1 ELSE 0 END) as isBought
FROM song, musicObject mo, musicObjectGenre mog
WHERE song.song_id = @song_id and
song.song_id = mo.music_ibject_id and
buys.music_object_id = mo.music_object_id and
mog.music_object_id = mo.music_object_id
```

2) Getting similar songs:

```
WITH curGenre(value) as (SELECT genre_name
                         FROM musicObjectGenre mog
                         WHERE mog.music_object_id=@song_id)

SELECT name, artist_username, score
FROM song, musicObject mo, musicObjectGenre mog
WHERE song.song_id = mo.music_objecet_id and
mo.music_object_id = mog.music_object_id and
mog.genre_name = curGenre.value and
song.song_id  <> @song_id
ORDER BY score DESC
```

3) Getting comments:

```
SELECT parCom.desc as parentDesc, parCom.date as parDate,
parCom.username parUs, childCom.desc as childDecs,
childCom.date as childDate, childCom.username as childUs
FROM comment parCom LEFT OUTER JOIN replyTo
ON parCom.comment_id = replyTo.parent_comment_id JOIN
comment childCom ON
replyTo.reply_comment_id = childCom.comment_id
WHERE parCom.music_object_id = @song_id
```

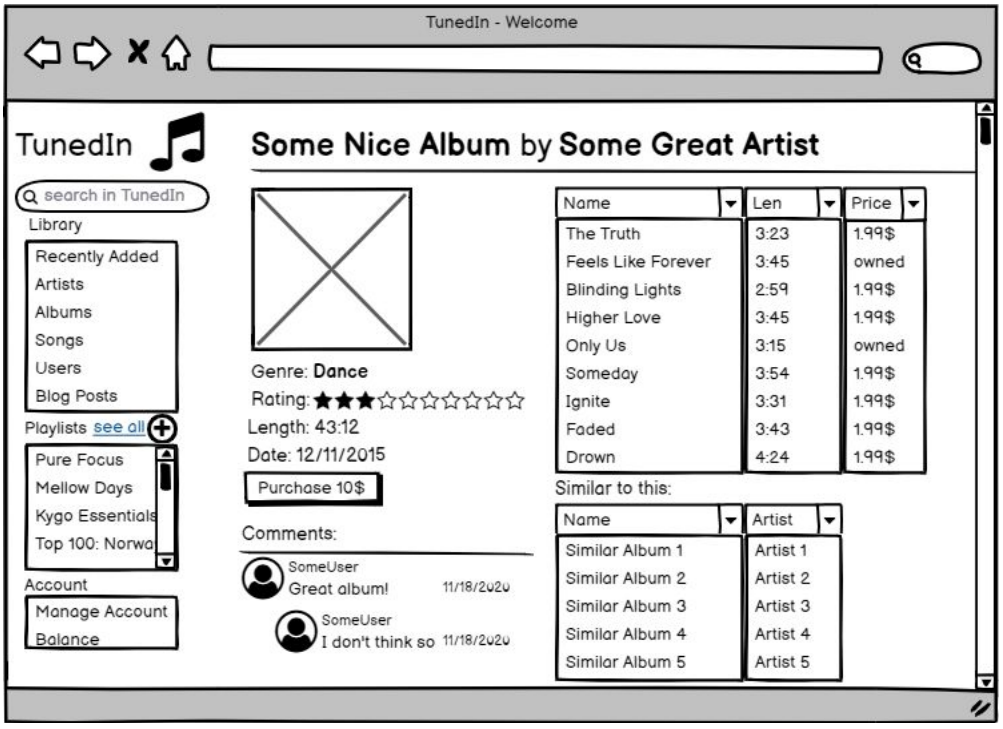
4) Adding a new comment:

```
INSERT INTO comment
values (@comment_desc, CURRENT_TIMESTAMP, @username,
null, @song_id);
```

5) Adding a new reply:

```
INSERT INTO comment
values (@comment_desc, CURRENT_TIMESTAMP, @username,
null, @song_id);

INSERT INTO replyTo
values (@parent_comment_id, LAST_INSERT_ID())
```

4.19. Album profile



Input: @album_id

Process: On the album profile page, the user can view the name, artist, genre, length, overall rating, the date of purchase, and the cover img of the album. Moreover, the user can see her rating of the album (if available) and the comments on the songs by other users. The user is also provided with a list of similar songs based on the genre ordered by the rating. If the user has not yet purchased the album, she can do so by clicking the purchase button. Alternatively, the user can buy individual songs from the album by clicking on the price of the album.

SQL Queries:

1) Getting basic album info:

```
SELECT mo.name as songName, mo.artistusername as artistName,
mo.genre as genre, length, release_date, score, cover_img,
price, (SELECT CASE WHEN EXISTS(
                                SELECT *
                                FROM buys
                                WHERE username = @username and
                                music_object_id = @album_id
                                ) THEN 1 ELSE 0 END) as isBought
FROM album, musicObject mo, musicObjectGenre mog
WHERE album.album_id = @album_id and
album.album_id = mo.music_ibject_id and
mog.music_object_id = mo.music_object_id
```

2) Getting songs in the album:

```
SELECT name, length, price, number,
        (SELECT CASE WHEN EXISTS(
                        SELECT *
                        FROM buys
                        WHERE username = @username and
                        music_object_id = @album_id
                        ) THEN 1 ELSE 0 END) as isBought
FROM song, musicObject mo
WHERE song.album_id = mo.music_object_id;
```

3) Getting similar albums:

Similar to getting similar songs

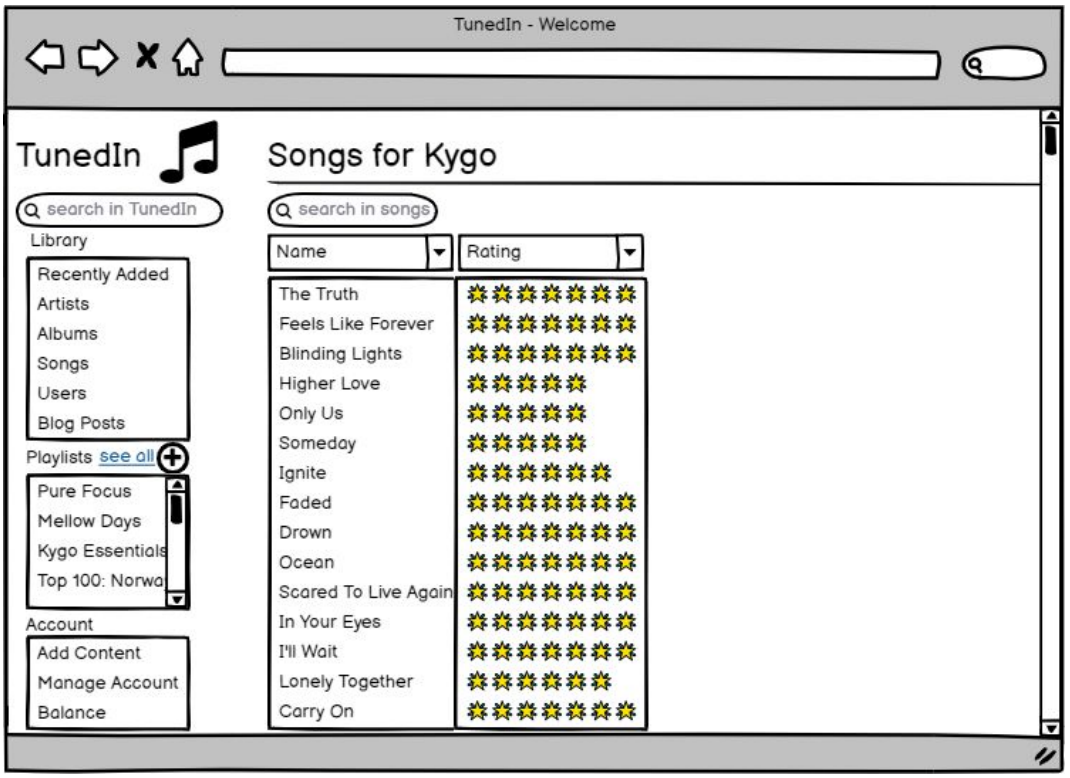
4) Getting comments:

Similar to getting comments for the song profile

5) Buying an album (buying a song is similar):

```
INSERT INTO buys
values(@username, @album_id, CURRENT_TIMESTAMP);
```

4.20. Artist songs



Input: None

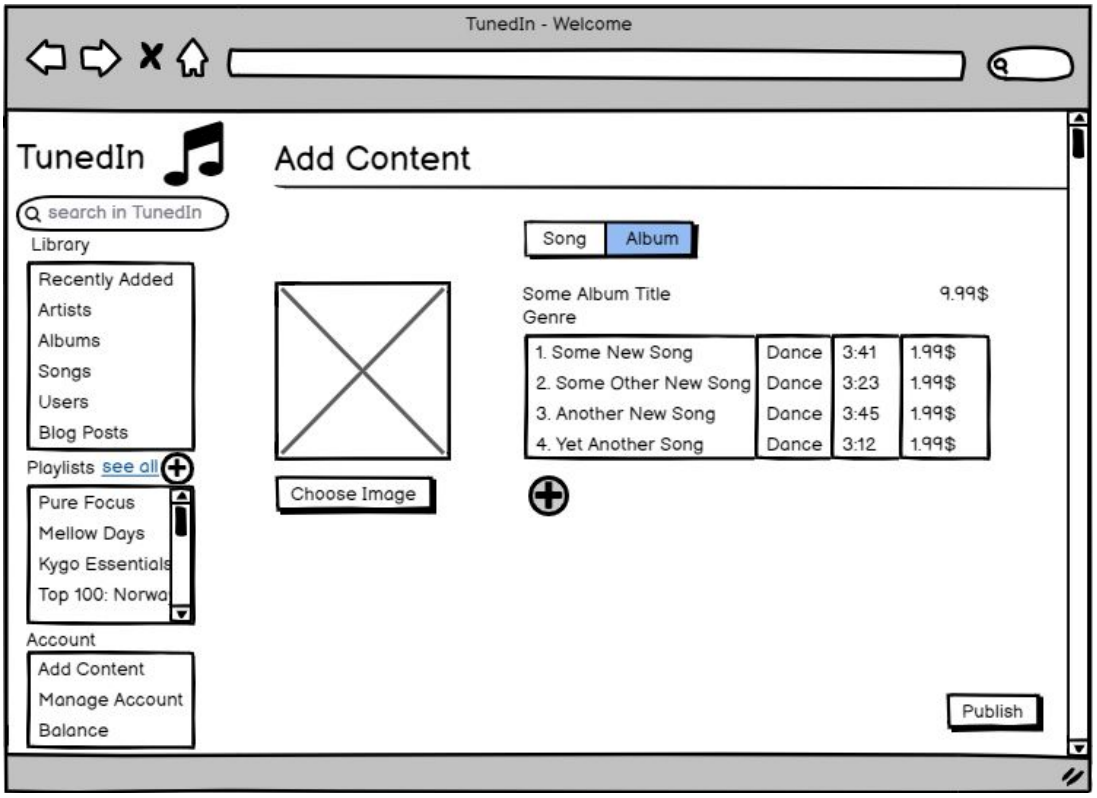
Process: The artist can view a list of her songs ordered by scores. The artist can change the order by pressing the arrow corresponding to the name of the column.

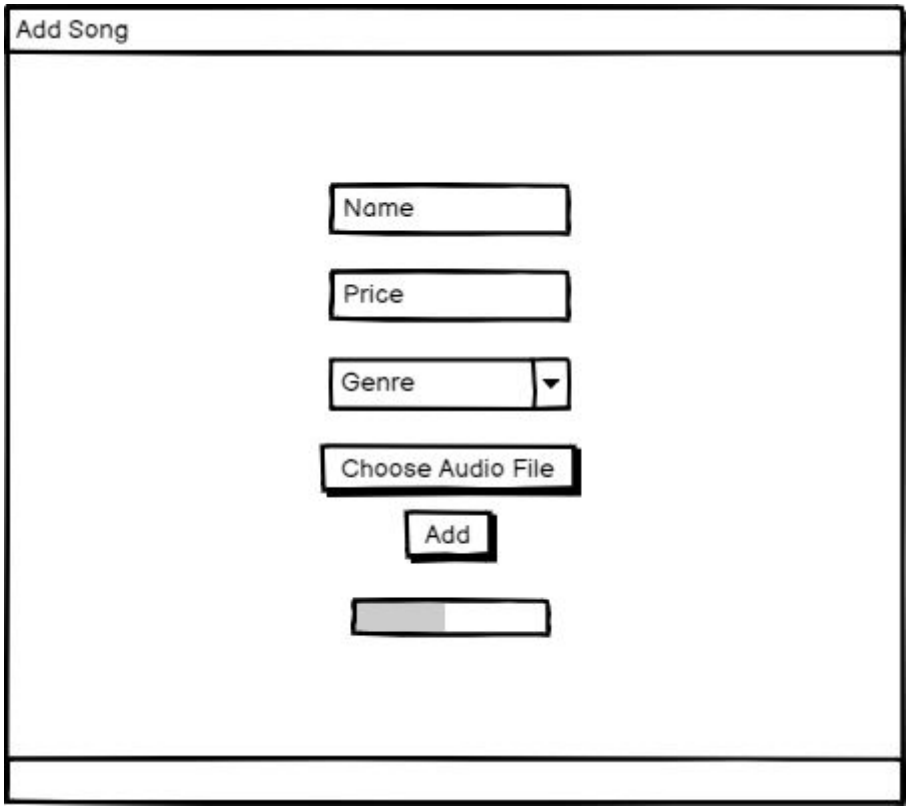
SQL Queries:

1) Getting the list of songs:

```
SELECT name, score
FROM song, musicObject mo
WHERE song.song_id = mo.music_object_id and
mo.artist_username = @artist_username
ORDER BY score DESC;
```

4.21. Artist add album & add song





Inputs: @album_name, @album_price, @album_genre, @cover_img, @song_name, @song_price, @song_genre

Process: The artist can upload a new album on this page. The artist is able to input the title of the album, cover_img, genre, and price. Pressing the '+' button redirects the artist to a page where she can insert songs to the album by setting the name, price, and genre.

SQL Queries:

1) Uploading an album:

```
INSERT INTO musicObject(name, price, release_date, cover_img, artist_username)
values(@album_name, @album_price, CURRENT_TIMESTAMP, @cover_img, @artist_username);

INSERT INTO musicObjectGenre
values(LAST_INSERT_ID(), @album_genre);
```

2) Uploading a song (@album_id corresponds to the current album):

```
INSERT INTO musicObject(name, price, release_date, artist_username)
values(@song_name, @song_price, CURRENT_TIMESTAMP, @cover_img, @artist_username);

INSERT INTO song (song_id, album_id)
values (LAST_INSERT_ID(),  @album_id);
```

5. Website

The report is available at https://kizilkayaarda.github.io/CS353/.