

VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



DATABASE SYSTEM

Assignment 2

Free Music Streaming Service

Advisors:	Tran Minh Quang	
	Dang Tran Khanh	
Candidates:	Tran Hoang Long	– 1852545
	Pham Hoang Hai	– 1852020
	Tran Quoc Anh	– 1852247

Ho Chi Minh City, April 7, 2021



Contents

1	Introduction	2
2	Design	2
2.1	Requirement Description	2
2.1.1	Data Requirements	2
2.1.2	Functional Requirements	2
2.2	Conceptual Database Design	3
2.3	Logical Database Design	5
3	Implementation	6
3.1	DBMS Choice	6
3.2	Logical Implementation	6
3.3	Result	9
4	Database Normalization	13
4.1	First Normal Form	13
4.2	Second Normal Form	13
4.3	Third Normal Form and BCNF	15
5	Database Manipulation	15
5.1	Query	15
5.1.1	Recording - Track - Album Queries	15
5.1.2	Playlist - User - User Queue Queries	16
5.1.3	Other Queries	17
5.1.4	Lesser Important Queries	17
5.2	Trigger	18
5.3	Store Procedures	18
5.4	Indexing	19
5.4.1	Auto-Index	19
5.4.2	Manual-Index	21
6	Application Layer	26
6.1	Music Browsing	26
6.2	Admin Interface	31
6.3	Searching Feature	31
7	Security Improvements	32
7.1	Authorization and Role Assignment	32
7.2	SQL Injection	33
8	Conclusion	34
9	Log	34

1 Introduction

With advances in data compression, music streaming become more practical and more popular. The 2000s witnessed the birth of many popular music streaming service like Spotify, SoundCloud and many other. With increasing traffic from new users, these services need fast and efficient database systems. In this assignment, we will design a simple database schema for a free music streaming service.

2 Design

2.1 Requirement Description

Purpose

Designing a database system for a free music streaming service.

2.1.1 Data Requirements

1. A recording is the unit of music storage. This represents the digital copy of a song. Represented by its ID, name and duration. We will refer to a recording as a **song**.
2. Though recordings are the unit of storage, in our music service, songs are presented to listener (normal user) in the form of tracks, which is the appearance of a song in a specific album. A track holds information about its resource (recording ID) as well as a track number representing its position in a specific album. We will refer to tracks as **available songs**.
3. In real world practice, an artist will not release songs themselves but rather wrap songs in an album, even single song can be wrapped in single album. Here, we try to represent that interaction in our database, album is the container for available songs that we present to our listener. An album has its own id, name, release date and type. Album type can be Single, EP or Album.
4. Of course, album must be owned by one or many artists.
5. Every song must be credited by some artist. A credit associates an artist with their role in the making of a song. Artist's role can be Writer, Performer or Producer.
6. User access our system by registering and logging in to their assigned role. User's role can be Listener, Artist, Moderator or System Admin. Information of a user includes their user id, username, password, role and their **music queue**.
7. A user has their own music queue representing their playback status such as: repeat state (repeat once, no repeat, repeat all), play or paused, the information of current song in queue being played (index) and the current process of playing that song (how many seconds of the song already played). Tracks are added to user's queue and before being played.
8. Besides browsing album for available songs to play, user can also create their own playlist and add in available songs. These songs are ordered by the time which they are added. User can set their playlists to public or private.

2.1.2 Functional Requirements

We will specify the features/transactions that are available for different user roles in our system, including Listener, Artist, Moderator, Administrator.

Listener

1. Browse for available songs (track) from albums and playlists (public or of their own)
2. Browse for available songs that is owned by or credit a specific artist
3. User can browse back and forth between related songs/album/playlist/artist
4. See detailed information of available song/album/playlist/artist/user profile/playing queue status.
5. User can select track to be added to their play queue
6. User can create their own playlist. They can also public their playlist to share with others.
7. User can add track to their playlist
8. Search for artists , available songs, albums, public playlists by name and also filter their result:
 - sort song by duration
 - sort album by type
 - sort search result ascending/ descending.
9. Of course, user can change their information such as password, playlist name

Artist

1. Artist has all the functionalities of a user with tracks.
2. Artist can also change information about albums/recordings/tracks that they own.

Administrator

1. Direct access to all aspects of the database
2. The only person that can add/remove other users

2.2 Conceptual Database Design

Derived from the above gathered requirements, this is the conceptual design of our database. Some parts of the diagram will be explained in more details.

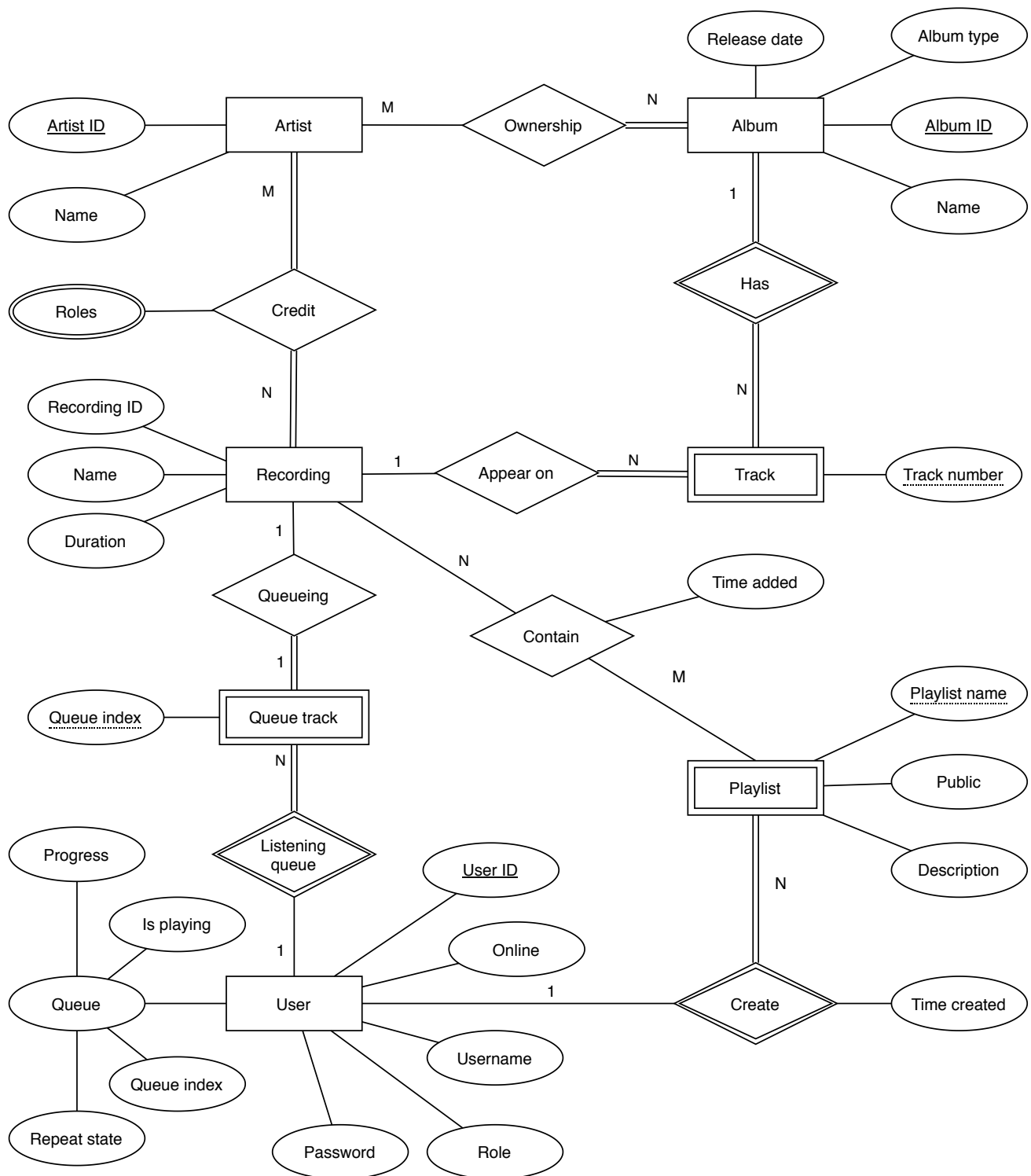


Figure 1: ER-Diagram Of Conceptual Design

Recording This is the most important entity which can be linked to tracks on albums. A recording has a name and duration, identified by its ID.

Playlist A playlist is created by an user and is private to that user (unless set to public), so the weak

Playlist entity and its partial key *Playlist name* means that there can be many playlist with the same name in our system, but for one user, their playlists must have non-duplicate name. So a playlist is identified through its name and user ID.

Listening queue and Queue track *Listening queue* represents an user's queue for each *User*, and *Queue track* represents tracks that are waiting in some queue. Every track that is queued must have their own *Queue index*, combined with their *User ID* to define whose queue the track is in.

Ownership This is the relationship of an *Artist* having the owner right to an *Album*. It can be seen than all albums must have owner.

Artist credit The *Credit* relationship shows that an artist must participate in making a recording, in whatever role, or even multiple roles. And on the other hand, a recording must have credit (well of course that a recording must be made by someone).

Artist–Album–Track Looking at the relation between these 3 entity, we must have some observations, all regarding to our requirements:

- Album must be owned and cannot be empty
- A track is always in an album and is defined by the album that contains it.
- A track must be made by some one (some artist)
- An artist must at least participate in making a track, but is not entitled to own any album.

2.3 Logical Database Design

Using mapping strategies, we have converted the conceptual design into logical design as follows:

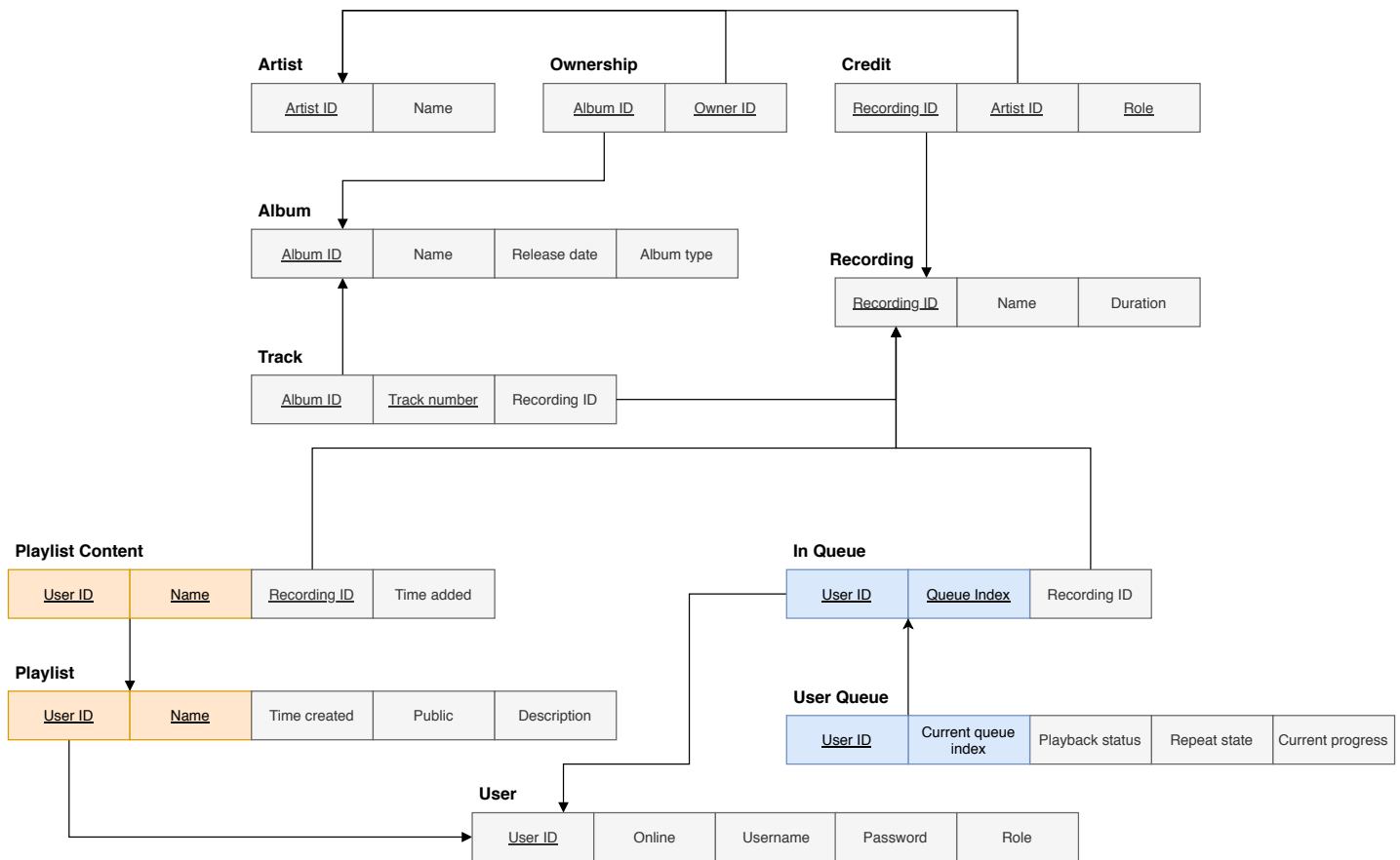


Figure 2: Logical Design of Database

Note:

1. Highlighted composite keys: Composite keys in our diagram is highlighted for better clarity.
2. Participation constraint: Participation constraint such as **Total participation** is not shown in the logical design and will be implemented in later phases.

3 Implementation

3.1 DBMS Choice

For this assignment, we implement the database using SQLite. What set it apart from other DBMS is that SQLite is not a client-server database engine but rather it is embedded into the end program. Therefore, it is very easy to set up and is appropriate for applications that do not have much traffic.

Furthermore, unlike most SQL database engines that use static, rigid typing, SQLite uses a more general dynamic type system. In SQLite, the datatype of a value is associated with the value itself, not with its container. Thus it is very easy to add new data types that are not in SQL standard.

3.2 Logical Implementation

Artist Table

For the `artist` table, primary key will be attribute `artist_id` of type integer. And the name of the artist will be text and must not be NULL.

```
CREATE TABLE IF NOT EXISTS artist(  
    artist_id INTEGER NOT NULL PRIMARY KEY,  
    artist_name TEXT NOT NULL  
);
```

Recording Table

Recording is defined by its `recording_id` and `recording_name`. Note that `duration` is positive integer (second)

```
CREATE TABLE IF NOT EXISTS recording(  
    recording_id INTEGER NOT NULL PRIMARY KEY,  
    recording_name TEXT NOT NULL,  
    duration INT NOT NULL CHECK (duration > 0)  
);
```

Track Table

The `track` table references to `album` table to show the relation between Track and Album (track must be in album). Attribute `track_number` starts at 1 and `track_duration` must be positive.

```
CREATE TABLE IF NOT EXISTS track(  
    album_id NOT NULL REFERENCES album(album_id) ON DELETE CASCADE,  
    recording_id NOT NULL REFERENCES recording(recording_id) ON DELETE CASCADE,  
    track_number INT NOT NULL CHECK (track_number >= 1),  
    PRIMARY KEY (album_id, track_number)  
);
```

Album Table

The `album` table will contain information about album. The `album_id` and `album_name` attributes are the same as `artist` table, with two more nullable attributes `release_date` and `album_type`.

```
CREATE TABLE IF NOT EXISTS album(  
    album_id INTEGER NOT NULL PRIMARY KEY,  
    album_name TEXT NOT NULL,  
    release_date DATE NOT NULL,  
    album_type ALBUM_TYPE NOT NULL  
);
```

Ownership Table

The `ownership` table holds references to `album_id` and `artist_id` showing relation between them.

Note: The use of `ON DELETE CASCADE` here means when the parent entity is deleted, the child entity must also be.

```
CREATE TABLE IF NOT EXISTS ownership(  
    album_id NOT NULL REFERENCES album(album_id) ON DELETE CASCADE,  
    artist_id NOT NULL REFERENCES artist(artist_id) ON DELETE CASCADE,  
    PRIMARY KEY (album_id, artist_id)  
);
```

Credit Table

The `credit` table store artist's credit on a track.


```
CREATE TABLE IF NOT EXISTS credit(  
    recording_id INT NOT NULL REFERENCES recording(recording_id) ON DELETE CASCADE,  
    artist_id INT NOT NULL REFERENCES artist(artist_id) ON DELETE CASCADE,  
    role CREDIT_ROLE NOT NULL,  
    PRIMARY KEY (recording_id, artist_id, role)  
);
```

User Table

The user table store user identification and authorization

- username and password used for authentication and username must be UNIQUE
- online is user's status

```
CREATE TABLE IF NOT EXISTS user(  
    user_id INTEGER NOT NULL PRIMARY KEY,  
    online BOOL NOT NULL,  
    username text not NULL UNIQUE,  
    password text not NULL,  
    role USER_ROLE NOT NULL  
);
```

User Queue Table

The user_queue table store user playback status, settings.

- repeat_state: user options for song repetition (repeat one song, play through, repeat all songs)
- is_playing: the playback status (play/paused)
- current_queue_idx: the current index of the song being played from user's listening queue
- cur_progress: progress into the currently playing track (elapsed time in seconds)

```
CREATE TABLE IF NOT EXISTS user_queue(  
    user_id INT NOT NULL PRIMARY KEY,  
    repeat_state REPEAT_STATE NOT NULL,  
    is_playing BOOL NOT NULL,  
    cur_queue_idx INT NOT NULL CHECK (cur_queue_idx >= 0),  
    cur_progress INT NOT NULL CHECK (cur_progress >= 0),  
    FOREIGN KEY (user_id, cur_queue_idx) REFERENCES in_queue(user_id, queue_index)  
    ON DELETE CASCADE  
);
```

Playlist Table

The playlist store information of all user-created tables, including its created time and publicity.

```
CREATE TABLE IF NOT EXISTS playlist(  
    user_id INT NOT NULL REFERENCES user(user_id) ON DELETE CASCADE,  
    playlist_name TEXT NOT NULL,  
    description TEXT NOT NULL,  
    time_created TIMESTAMP NOT NULL,  
    is_public INT NOT NULL,  
    PRIMARY KEY (user_id, playlist_name)  
);
```

Playlist Content Table

The `playlist_content` table store the tracks included in a playlist as well as their added time.

```
CREATE TABLE IF NOT EXISTS playlist_content(  
    user_id INT NOT NULL,  
    playlist_name TEXT NOT NULL,  
    recording_id INT NOT NULL REFERENCES recording(recording_id) ON DELETE CASCADE,  
    time_added TIMESTAMP NOT NULL,  
    FOREIGN KEY (user_id, playlist_name) REFERENCES playlist(user_id, playlist_name)  
    ON DELETE CASCADE ON UPDATE CASCADE,  
    PRIMARY KEY (user_id, playlist_name, recording_id)  
);
```

In Queue Table

The `in_queue` table store recordings that are queued for an user. Each recording has a `queue_index` (non-negative).

```
CREATE TABLE IF NOT EXISTS in_queue(  
    user_id INT NOT NULL REFERENCES user(user_id) ON DELETE CASCADE,  
    recording_id INT NOT NULL REFERENCES recording(recording_id) ON DELETE CASCADE,  
    queue_index INT NOT NULL CHECK (queue_index >= 0),  
    PRIMARY KEY (user_id, queue_index)  
);
```

3.3 Result

With the logical implementation above, we have created a new database for our music streaming service including 9 tables. In SQLite, we use the command `.tables` to list all tables in the database. SQLite can be integrated with Python using `sqlite3` module. Using it will save a lot of time instead of typing command one by one. Here, we will use it to insert many new records with input values at one go. Every attributes of table must be passed through using Python's parameter substitution except primary key which is generated automatically.

```
INSERT INTO album(name,release_date,album_type)  
VALUES(?,?,?);
```

Do the same with the rest, but it must follow the rules that we have set in the logical implementation else we will get `FOREIGN KEY constraint failed` error, which means we have passed a duplicate primary key or a non-existent key.

SQLite version 3.33.0 2020-08-14 13:23:32

Enter ".help" for usage hints.

sqlite> .tables

album	in_queue	playlist_content	user
artist	ownership	recording	user_queue
credit	playlist	track	

sqlite> .modes column

sqlite> selSELECT * FROM album;

album_id	album_name	release_date	album_type
1	Double trouble	1907-02-14	0
2	No comment	1930-03-05	1
3	Preaching choir	1938-03-18	2
4	Ice cold	1979-03-28	0

5	Sleeping dogs	1969-07-16	1
6	Blank canvas	1986-12-02	2
7	Concept art	2010-03-09	0
8	No direction	2017-12-06	1
9	Grains of salt	2019-03-27	2
10	Curves	2019-11-28	0

sqlite> SELECT * FROM artist;

artist_id	artist_name
-----------	-------------

1	Hunter Marsh
2	Karl Laine
3	Thorin Dikan
4	Bobby Dell
5	Erik Elliot
6	River Alexander
7	Alexis Arthur
8	Logan Raine
9	Lee Lord
10	Haiden Baker

sqlite> SELECT * FROM credit;

recording_id	artist_id	role
--------------	-----------	------

1	1	2
2	2	2
3	3	2
4	4	2
5	5	2
6	6	2
7	7	2
8	8	2
9	9	2
10	10	0
1	3	0
1	3	1
2	4	1
3	1	0
4	10	1
5	10	0
5	10	1
5	10	2
6	7	2
7	4	1
8	5	1
9	5	2
10	2	0

sqlite> SELECT * FROM in_queue;

user_id	recording_id	queue_index
---------	--------------	-------------

11	5	1
11	8	2
11	15	3
12	2	1
12	10	2
13	11	1



13	9	2
15	1	1
15	3	2
15	12	3
15	8	4
14	11	1
14	7	2
16	16	1
16	18	2
16	20	3
16	1	4
16	2	5

sqlite> SELECT * FROM ownership;

album_id	artist_id
----------	-----------

1	3
1	4
2	1
3	5
3	6
3	1
4	3
4	4
5	5
5	6
6	7
6	1
6	2
7	1
7	2
7	4
8	5
9	1
10	2
10	7

sqlite> SELECT * FROM playlist;

user_id	playlist_name	description	time_created	is_public
2	Chill	Music to chill	2000-11-22	0
4	boringgg	sad music	2015-05-13	0
4	romance and pop	send this to crush	2020-11-12	1
8	lofi	study and work music	1999-06-26	0
5	battle rap	my favor diss track	2005-01-02	1
5	bolero	music for my mom and dad	2001-09-23	1
2	bolero	music for my mom and dad	2001-09-23	1
5	trap,edm,remix	quay? len anh em oi	2019-9-19	1
9	bts trash	no girl here, no trigger	2011-06-09	0

sqlite> SELECT * FROM playlist_content;

user_id	playlist_name	recording_id	time_added
2	Chill	5	2000-11-22
2	Chill	12	2000-11-23
2	Chill	10	2000-11-24
4	boringgg	7	2015-05-13



4	romance and pop	1	2020-11-12
4	romance and pop	13	2020-12-01
5	battle rap	8	2005-01-02
5	battle rap	15	2005-10-12
5	bolero	2	2001-09-23
5	bolero	3	2001-09-25
5	bolero	6	2001-10-02
5	bolero	13	2002-10-11
5	bolero	14	2002-10-15
5	trap,edm,remix	17	2019-09-19
5	trap,edm,remix	15	2019-09-25
8	lofi	4	1999-06-26
8	lofi	9	1999-06-27
9	bts trash	10	2011-06-09
9	bts trash	11	2011-06-14
9	bts trash	20	2011-06-16

sqlite> SELECT * FROM recording;

recording_id	recording_name	duration
1	Call	295
2	Sunshine, My Choice	430
3	Hibernate Magic	341
4	Cold Era	500
5	Closer Earth	234
6	Instrumental Heaven	266
7	Carebbian Saloon	421
8	Unused Home	332
9	The Hottest Soul	267
10	#That Violin	438

sqlite> SELECT * FROM track;

album_id	recording_id	track_number
1	8	1
1	9	2
1	10	3
1	4	4
2	5	1
2	5	2
3	8	1
3	6	2
4	7	1
5	4	1
6	5	1
7	6	1
7	7	2
8	8	1
8	5	2
8	10	3
9	5	9
9	5	15
10	3	45
10	3	200

sqlite> SELECT * FROM user;

user_id	online	username	password	role
---------	--------	----------	----------	------

```

-----
1      0      weirdkid      123abc      2
2      0      bigtimegangsta 987654sad  2
3      0      pphai         haizzzzzzzz 3
4      0      someone       abcdefg      0
5      0      daddypung      rememberthis 1
6      0      whatthehell    mypassword   0
7      0      kid_vippro     notvipproanymore 3
8      0      pplfuckme      sohard       3
9      0      sotired        healthynbalance 0
10     0      something      helloworld   1
11     1      online1        123456789    0
12     1      online2        123123123    3
13     1      online3        111111111    3
14     1      online4        555555555    0
15     1      online5        987654321    1
16     1      online6        000000000    1
sqlite> SELECT * FROM user_queue;
user_id  repeat_state  is_playing  cur_queue_idx  cur_progress
-----
11       2              1           2             250
12       1              0           2             432
13       1              1           1             50
14       0              1           1             83
15       0              0           3             62
16       2              1           5            222

```

4 Database Normalization

In this section, we will try to improve our database schema using normalization

4.1 First Normal Form

The first normal form can be seen as a formal definition of a basic relational model rather than an improvement on certain ambiguous aspects of database schema. As defined, 1NF does not allow set of values or tuple of values as an attribute field within a tuple. The only attribute values allowed are single **atomic** values.

Considering our database for the 1NF test, we can see from our conceptual design in Figure 1, specifically in relation **User**, that we have nested attributes, namely **Queue** and **Authorization**. But moving on to the logical design phase and through mapping algorithm, our database logical implementation (Figure 2) does not have redundant multi-valued attributes. Both **Queue** and **Authentication** has been decomposed into their own table. Thus passing the 1NF test.

4.2 Second Normal Form

The second normal form disallows partial dependency of all non-prime attributes on all keys in the relation. All composite keys have to be tested to confirm full functional dependency.

To perform the test, we take a look through all of our relations' functional dependency in Figure 3, looking for redundant composite keys. Our composite keys includes:

1. {Album Id, Track Number} in Track Relation: Album Id specifies which album a track belongs to and track number is the track's "position" in that album. Neither of these can uniquely define a track or any of it's attribute
2. {User Id, Name} in Playlist Relation: A playlist does not have unique name because playlists are essentially public and is only unique to each individual user. No partial dependency here.

3. {User Id, Name, Album Id, Track Number} in Playlist Content Relation: In this relation, there's only one non-prime attribute that's **Time Added** and this is only defined by the primary key, not by any of its components.

4. {Album Id, Track Number, Artist Id, Role} in Credit Relation: This relation is just a primary key.

After those specifications, it can be concluded that none of our keys violate the full functional dependency rule. Thus no decomposition is needed and our DB is already in 2NF.

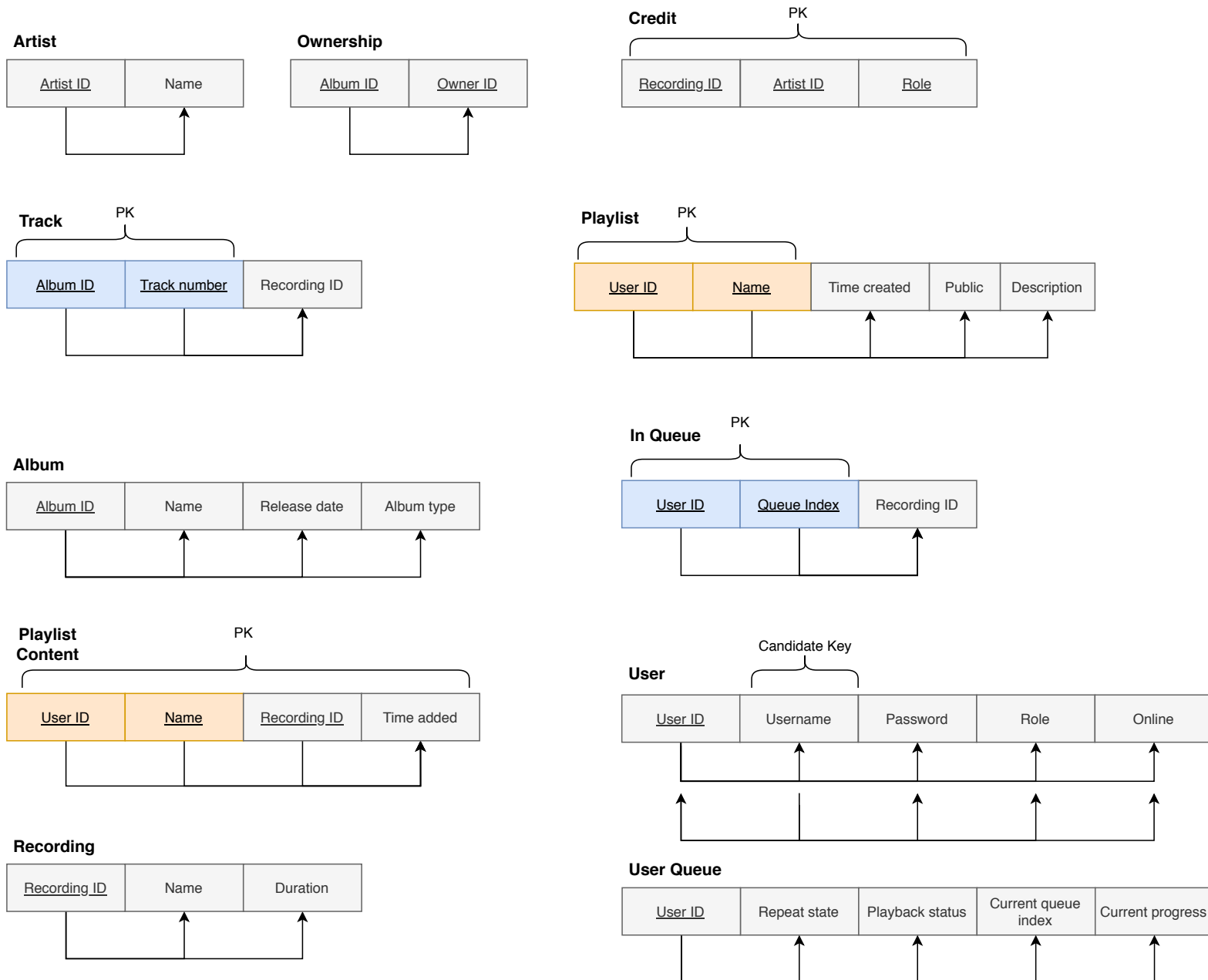


Figure 3: Functional Dependency of all Relations in the Database

4.3 Third Normal Form and BCNF

The last stop for normalization of our database is the third normal form and its extension Boyce-Codd normal form (BCNF). We do not discuss higher normal forms due to complexity and possible performance compromise.

As BCNF is stricter and includes 3NF, we can discuss BCNF instead. It focuses on removing trivial (transparent) functional dependency and only allow trivial dependency on the **super key** of the relation.

Going back to our dependencies in Figure 3, it's clear that there's no transitive functional dependencies in our relations. Thus putting it in BCNF (passing 3NF also).

5 Database Manipulation

5.1 Query

In this section, i will discuss some queries relating to our database. Lesser meaningful or lesser complex queries will be in their own sub-section (sub-section 5.1.4).

5.1.1 Recording - Track - Album Queries

Recordings that are not included as track in any albums

Recordings can be stored in our database and optionally included in albums as track. We can find out which recordings are not being used as tracks by getting the difference between recording table and track table.

```
SELECT recording_id, recording_name FROM recording
EXCEPT
SELECT recording_id, recording_name
FROM (recording NATURAL JOIN track)
```

Songs without a writer

This can be found by removing recordings that has writer credit from all recordings

```
SELECT recording_name, recording_id
FROM recording
EXCEPT
SELECT recording_name, recording_id
FROM
    recording
    NATURAL JOIN credit
WHERE role = 2
```

Track count of all albums

Counting album tracks is quite common and easy to query.

```
SELECT album_name , COUNT(recording_id) as track_count
FROM
    album
    NATURAL JOIN track
    NATURAL JOIN recording
GROUP BY album_id
```


Search by name fragment all tracks and albums

A common searching query is to search in albums and tracks for a specific name keyword. Here we use the LIKE keyword to implement this search.

Ex: Search for keyword 'me' in all album and track names. Note that in this example, the search is not whole word match and also not case sensitive.

```
PRAGMA case_sensitive_like = false;
SELECT recording_name, album_name
FROM
    recording
    NATURAL JOIN track
    NATURAL JOIN album
WHERE recording_name like '%me%' or album_name LIKE '%me%'
```

Albums with non-contiguous tracklist

In our albums, each track has their own track number representing its position in the album. But when tracks are added, there could be 'holes' in the tracklist where the track numbers are not contiguous. This can be spotted if the maximum track number is larger than the total number of tracks (knowing that track number is greater than 0)

```
SELECT album_name
FROM
    album
    NATURAL JOIN track
GROUP by album_id
HAVING MAX(track_number) > COUNT(track_number)
```

Albums with duplicated recordings

In our implementation, an album can have many tracks of the same recording, as long as they have different track number. This can be checked by joining tracks to themselves and compare recording id if tracks in the same album that has different track number.

```
SELECT DISTINCT(album_name)
FROM
    (track track1
    LEFT JOIN track track2
    on track1.album_id = track2.album_id AND track1.track_number != track2.track_number)
    NATURAL join album
WHERE track1.recording_id = track2.recording_id
```

5.1.2 Playlist - User - User Queue Queries

Trending songs

A song is considered "trendy" when lots of users are currently listening to that song. To obtain top 5 trending songs, we need to join 3 table `in_queue`, `user_queue` and `recording`.

```
SELECT COUNT(*) AS count, recording.recording_name
FROM in_queue JOIN user_queue
ON in_queue.queue_index = user_queue.cur_queue_idx
AND in_queue.user_id = user_queue.user_id
NATURAL JOIN recording
GROUP BY recording_id
ORDER BY count DESC
LIMIT 5;
```

Hottest songs

“Hotness” of a song is defined by the number of playlist containing that song. To obtain top 5 hottest song, the query is almost the same as above.

```
SELECT COUNT(*) AS count, recording.recording_name
FROM playlist_content NATURAL JOIN recording
GROUP BY recording_id
ORDER BY listening_count DESC
LIMIT 5;
```

5.1.3 Other Queries

5.1.4 Lesser Important Queries

Top 5 newest albums

```
SELECT album_name from album ORDER by release_date DESC LIMIT 5
```

Tracks sorted out by album type and duration

```
SELECT recording_name
FROM
    track
    NATURAL JOIN recording
    NATURAL JOIN album
WHERE album_type = 1 and duration > 300
```

Top most credited artist

```
SELECT artist_name, COUNT( DISTINCT role) as credit_count
FROM
    (artist
    NATURAL JOIN credit)
GROUP BY artist_id
```

Artist with credit on all 3 roles

```
SELECT artist_name, COUNT( DISTINCT role) as credit_count
FROM
    (artist
    NATURAL JOIN credit)
GROUP BY artist_id
HAVING credit_count = 3
```

5.2 Trigger

Censor bad words

Swear and profanity is considered unappropriated in professional world. Therefore, we will censor some bad words on recording name using star symbol.

```
CREATE TRIGGER censor_badword
AFTER INSERT ON recording
WHEN NEW.recording_name GLOB '*fuck*'
BEGIN
    UPDATE recording
    SET recording_name = replace(NEW.recording_name, 'fuck', 'f**k')
    WHERE recording_id = NEW.recording_id;
END;
```

Clean up empty album

Artists are allow to delete tracks in albums that they own, so naturally there will be some empty albums. When an artist delete the last track from an album, a trigger will be automatically fired to delete the album from the database.

```
CREATE TRIGGER cleanup_album
AFTER DELETE ON track
WHEN NOT EXISTS (
    SELECT *
    FROM track
    WHERE track.album_id = OLD.album_id
)
BEGIN
    DELETE FROM album
    WHERE album.album_id = OLD.album_id;
END;
```

5.3 Store Procedures

Unfortunately, SQLite does not support stored procedure so here we will write some stored procedure in standard SQL syntax for the sake of demonstration but they will be later implemented as Python functions.

All albums crediting an artist as a specific role

We usually look for songs performed by an artist or songs written by one. Here we simply query the according tables upon the condition of artist name and the needed role. Of course searching by name can yield songs from many artists.

```
CREATE PROCEDURE artist_role @artist_name TEXT @role CREDIT_ROLE
AS
SELECT DISTINCT recording_name
FROM
    artist
    NATURAL JOIN credit
    NATURAL JOIN recording
WHERE artist_name = @artist_name and role = @role
```

Artists that are credited in a recording but does not own any albums related to that recording

An interesting query is about artists credited in a recording but does not own any albums that contain tracks of that recording. To get result, we get the difference between artists credited in the recording and artists that owns the albums related.

```
CREATE PROCEDURE f @song_name TEXT
AS
    -- all artists credited
SELECT DISTINCT artist_name
FROM
    recording
    NATURAL JOIN credit
    NATURAL JOIN artist
WHERE recording_name = @song_name
EXCEPT -- minus
    -- artist that own rec's albums
SELECT DISTINCT artist_name
FROM
    recording
    NATURAL JOIN track
    NATURAL JOIN album
    NATURAL JOIN ownership
    NATURAL JOIN artist
WHERE recording_name = @song_name
```

Visible playlists

Playlist can be public or private, so when a user is browsing, they only see their private playlists and all other public ones.

```
CREATE PROCEDURE visible_playlists @user_id TEXT
AS
SELECT playlist_name
FROM playlist
WHERE is_public == 1
OR user_id = @user_id
```

5.4 Indexing

Indexing is a technique used for optimize querying from database by associating one (or many) additional look up structure called 'indexes'. This technique specifically helps with **SELECT** queries and **WHERE** clauses in SQL to retrieve data faster while slowing down **INSERT** and **UPDATE** due to the index needing to be updated as well.

We will be discussing pre-made (auto) indexes that's created by our framework and the indexes that we added in as well (manual).

5.4.1 Auto-Index

As for the our DBMS, SQLite, all the primary keys (atomic and composite) as well as unique fields are already indexed by the framework. This is the default behaviour of the DBMS to improve queries since querying on key values are always common.

To demonstrate the auto indexes available, we will see how the system react to some of our queries on two of our tables, namely **Playlist Content** and **User Table**.

User Table

Below queries uses the same index, this index contains the primary key of User table `user_id` and the unique field `username`.

Checking availability of username

```
EXPLAIN QUERY PLAN
SELECT *
FROM user
WHERE username = 'bigtimegangsta';
```

Result:

```
QUERY PLAN
|--SEARCH TABLE user USING INDEX sqlite_autoindex_user_1 (username=?)
```

Matching Username and Password

```
EXPLAIN QUERY PLAN
SELECT *
FROM user
WHERE username = 'trash' AND password = 'anothertrash';
```

Result:

```
QUERY PLAN
|--SEARCH TABLE user USING INDEX sqlite_autoindex_user_1 (username=?)
```

Playlist Content Table

All of the above queries below uses the auto index created for the Playlist Content Table, which includes `user_id`, `recording_id` and `playlist_name`.

All playlists of a user

```
EXPLAIN QUERY PLAN
SELECT *
FROM playlist_content
WHERE user_id = 3;
```

Result:

```
QUERY PLAN
|--SEARCH TABLE playlist_content USING INDEX sqlite_autoindex_playlist_content_1
↪ (user_id=?)
```

Find User's Playlist by Name

```
EXPLAIN QUERY PLAN
SELECT *
FROM playlist_content
WHERE user_id = 3 AND playlist_name = 'bolero';
```

Result:

```
QUERY PLAN
|--SEARCH TABLE playlist_content USING INDEX sqlite_autoindex_playlist_content_1
↪ (user_id=? AND playlist_name=?)
```

Get a specific song in a playlist

```
EXPLAIN QUERY PLAN
SELECT *
FROM playlist_content
WHERE user_id = 3 AND playlist_name = 'bolero' AND recording_id = 5;
```

Result:

```
QUERY PLAN
|--SEARCH TABLE playlist_content USING INDEX sqlite_autoindex_playlist_content_1
↪ (user_id=? AND playlist_name=? AND recording_id=?)
```

5.4.2 Manual-Index

For additional indexes, we can easily realize that our database has a very high demand use for searching items such as: recordings, albums, playlists, artists by name. So it's natural to add these needed indexes. Test for these indexes are included later.

Index Creation

Artist Name Index

```
CREATE INDEX IF NOT EXISTS artist_name_idx
ON artist(artist_name);
```

Recording Name Index

```
CREATE INDEX IF NOT EXISTS recording_name_idx
ON recording(recording_name);
```

Album Name Index

```
CREATE INDEX IF NOT EXISTS album_name_idx
ON album(album_name);
```

Playlist Name Index

```
CREATE INDEX IF NOT EXISTS playlist_name_idx
ON playlist(playlist_name);
```

Test Queries

Search Artist by name

```
EXPLAIN QUERY PLAN
SELECT *
FROM artist
WHERE artist_name = 'Thorin Dikan';
```

Result:

```
QUERY PLAN
`--SEARCH TABLE artist USING COVERING INDEX artist_name_idx (artist_name=?)
```

List all recordings that are not used as tracks

```
EXPLAIN QUERY PLAN
SELECT recording_id, recording_name
FROM recording
EXCEPT
SELECT recording_id, recording_name
FROM (recording NATURAL JOIN track);
```

Result:

```
QUERY PLAN
`--COMPOUND QUERY
  |--LEFT-MOST SUBQUERY
  | `--SCAN TABLE recording USING COVERING INDEX recording_name_idx
  `--EXCEPT USING TEMP B-TREE
    |--SCAN TABLE track
    `--SEARCH TABLE recording USING INTEGER PRIMARY KEY (rowid=?)
```

Search Album by name

```
EXPLAIN QUERY PLAN
SELECT *
FROM album
WHERE album_name = 'Curves';
```

Result:

```
QUERY PLAN
`--SEARCH TABLE album USING INDEX album_name_idx (album_name=?)
```

Playlists with same name

```
EXPLAIN QUERY PLAN
SELECT playlist_name
FROM playlist
GROUP BY playlist_name
HAVING COUNT(*) > 1;
```

Result:

```
QUERY PLAN
`--SCAN TABLE playlist USING COVERING INDEX playlist_name_idx
```

Tables (11)

Name	Type	Schema
album		CREATE TABLE album(album_id INTEGER NOT NULL PRIMARY KEY, album_name TEXT NOT NULL, release_date DATE NOT NULL, album_type ALBUM_TYPE NOT NULL)
album_id	INTEGER	"album_id" INTEGER NOT NULL
album_name	TEXT	"album_name" TEXT NOT NULL
release_date	DATE	"release_date" DATE NOT NULL
album_type	ALBUM_TYPE	"album_type" ALBUM_TYPE NOT NULL
artist		CREATE TABLE artist(artist_id INTEGER NOT NULL PRIMARY KEY, artist_name TEXT NOT NULL)
artist_id	INTEGER	"artist_id" INTEGER NOT NULL
artist_name	TEXT	"artist_name" TEXT NOT NULL
credit		CREATE TABLE credit(recording_id INT NOT NULL REFERENCES recording(id) ON DELETE CASCADE, artist_id INT NOT NULL REFERENCES artist(id) ON DELETE CASCADE, role CREDIT_ROLE NOT NULL, PRIMARY KEY (recording_id, artist_id, role))
recording_id	INT	"recording_id" INT NOT NULL
artist_id	INT	"artist_id" INT NOT NULL
role	CREDIT_ROLE	"role" CREDIT_ROLE NOT NULL
in_queue		CREATE TABLE in_queue(user_id INT NOT NULL REFERENCES user(user_id) ON DELETE CASCADE, recording_id INT NOT NULL REFERENCES recording(recording_id) ON DELETE CASCADE, queue_index INT NOT NULL CHECK (queue_index >= 0), PRIMARY KEY (user_id, queue_index))
user_id	INT	"user_id" INT NOT NULL
recording_id	INT	"recording_id" INT NOT NULL
queue_index	INT	"queue_index" INT NOT NULL CHECK("queue_index" >= 0)
ownership		CREATE TABLE ownership(album_id NOT NULL REFERENCES album(album_id) ON DELETE CASCADE, artist_id NOT NULL REFERENCES artist(artist_id) ON DELETE CASCADE, PRIMARY KEY (album_id, artist_id))
album_id		"album_id" NOT NULL
artist_id		"artist_id" NOT NULL
playlist		CREATE TABLE playlist(user_id INT NOT NULL REFERENCES user(user_id) ON DELETE CASCADE, playlist_name TEXT NOT NULL, description TEXT NOT NULL, time_created TIMESTAMP NOT NULL, is_public INT NOT NULL, PRIMARY KEY (user_id, playlist_name))
user_id	INT	"user_id" INT NOT NULL
playlist_name	TEXT	"playlist_name" TEXT NOT NULL
description	TEXT	"description" TEXT NOT NULL
time_created	TIMESTAMP	"time_created" TIMESTAMP NOT NULL
is_public	INT	"is_public" INT NOT NULL

Name	Type	Schema
playlist_content		CREATE TABLE playlist_content(user_id INT NOT NULL, playlist_name TEXT NOT NULL, recording_id INT NOT NULL REFERENCES recording(recording_id) ON DELETE CASCADE, time_added TIMESTAMP NOT NULL, FOREIGN KEY (user_id, playlist_name) REFERENCES playlist(user_id, playlist_name) ON DELETE CASCADE ON UPDATE CASCADE, PRIMARY KEY (user_id, playlist_name, recording_id))
user_id	INT	"user_id" INT NOT NULL
playlist_name	TEXT	"playlist_name" TEXT NOT NULL
recording_id	INT	"recording_id" INT NOT NULL
time_added	TIMESTAMP	"time_added" TIMESTAMP NOT NULL
recording		CREATE TABLE recording(recording_id INTEGER NOT NULL PRIMARY KEY, recording_name TEXT NOT NULL, duration INT NOT NULL CHECK (duration > 0))
recording_id	INTEGER	"recording_id" INTEGER NOT NULL
recording_name	TEXT	"recording_name" TEXT NOT NULL
duration	INT	"duration" INT NOT NULL CHECK("duration" > 0)
track		CREATE TABLE track(album_id NOT NULL REFERENCES album(album_id) ON DELETE CASCADE, recording_id NOT NULL REFERENCES recording(recording_id) ON DELETE CASCADE, track_number INT NOT NULL CHECK (track_number >= 1), PRIMARY KEY (album_id, track_number))
album_id		"album_id" NOT NULL
recording_id		"recording_id" NOT NULL
track_number	INT	"track_number" INT NOT NULL CHECK("track_number" >= 1)
user		CREATE TABLE user(user_id INTEGER NOT NULL PRIMARY KEY, online BOOL NOT NULL, username text not NULL UNIQUE, password text not NULL, role USER_ROLE NOT NULL)
user_id	INTEGER	"user_id" INTEGER NOT NULL
online	BOOL	"online" BOOL NOT NULL
username	text	"username" text NOT NULL UNIQUE
password	text	"password" text NOT NULL
role	USER_ROLE	"role" USER_ROLE NOT NULL
user_queue		CREATE TABLE user_queue(user_id INT NOT NULL PRIMARY KEY, repeat_state REPEAT_STATE NOT NULL, is_playing BOOL NOT NULL, cur_queue_idx INT NOT NULL CHECK (cur_queue_idx >= 0), cur_progress INT NOT NULL CHECK (cur_progress >= 0), FOREIGN KEY (user_id, cur_queue_idx) REFERENCES in_queue(user_id, queue_index) ON DELETE CASCADE)
user_id	INT	"user_id" INT NOT NULL
repeat_state	REPEAT_STATE	"repeat_state" REPEAT_STATE NOT NULL
is_playing	BOOL	"is_playing" BOOL NOT NULL
cur_queue_idx	INT	"cur_queue_idx" INT NOT NULL CHECK("cur_queue_idx" >= 0)
cur_progress	INT	"cur_progress" INT NOT NULL CHECK("cur_progress" >= 0)

Indices (4)

Name	Type	Schema
album_name_idx		CREATE INDEX album_name_idx on album(album_name)
album_name		"album_name"
artist_name_idx		CREATE INDEX artist_name_idx on artist(artist_name)
artist_name		"artist_name"
playlist_name_idx		CREATE INDEX playlist_name_idx on playlist(playlist_name)
playlist_name		"playlist_name"
recording_name_idx		CREATE INDEX recording_name_idx on recording(recording_name)
recording_name		"recording_name"

Views (0)

Name	Type	Schema
------	------	--------

Triggers (2)

Name	Type	Schema
censor_badword		CREATE TRIGGER censor_badword AFTER INSERT ON recording WHEN NEW.recording_name GLOB '*fuck*' BEGIN UPDATE recording SET recording_name = replace(NEW.recording_name, 'fuck', 'f**k') WHERE recording_id = NEW.recording_id; END
cleanup_album		CREATE TRIGGER cleanup_album AFTER DELETE ON track WHEN NOT EXISTS (SELECT * FROM track WHERE track.album_id = OLD.album_id) BEGIN DELETE FROM album WHERE album.album_id = OLD.album_id; END

6 Application Layer

For this section, we overlay our database with an application to demo some of our main features as well as capabilities. Our application is a web application implemented using Django framework.

Our focused feature in development of the app includes: Music Browsing, Playlist Management, Artist Interface, Admin Interface, Searching Feature and Security.

The part of security will be specifically discuss later. We mainly focus on the roles and functionalities here.

6.1 Music Browsing

The main purpose of our application is to allow our listeners to find their music and listen to what they prefer. A user that is a listener or artist will be greeted with the index home page when they login. This page includes the top/trendiest artists/tracks/albums/playlists on the network.

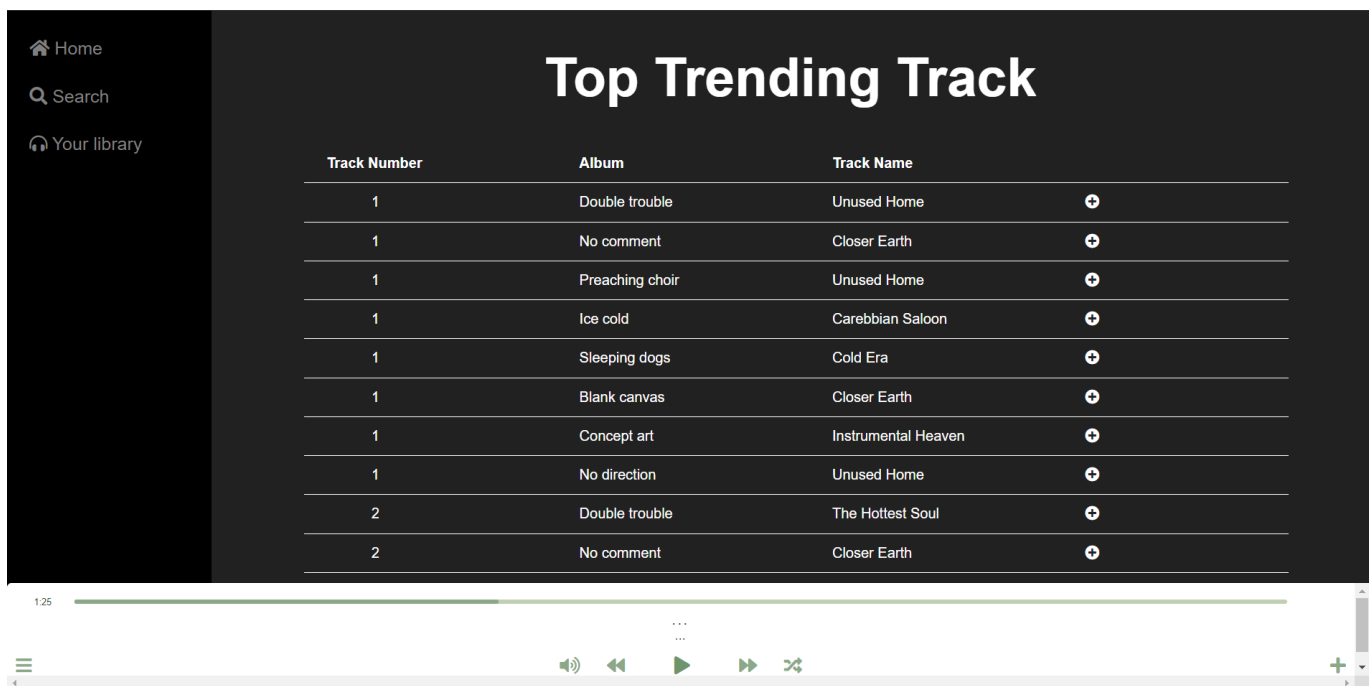



Figure 4: Index Page

User can select 'show more' to discover more of a specific type.


Lohaa Music

Search for Artists, Tracks or Albums...

Hello, tqa Logout

Home
Search
Your library

Latest Album

Name	Release Date	Owner	Type
Curves	Nov. 28, 2019	Alexis Arthur	Extended Play
Grains of salt	March 27, 2019	Hunter Marsh	Extended Play
No direction	Dec. 6, 2017	Erik Elliot	Single
Concept art	March 9, 2010	Karl Laine	Single
Blank canvas	Dec. 2, 1986	Logan Raine	Extended Play
Ice cold	March 28, 1979	Bobby Dell	Single
Sleeping dogs	July 16, 1969	River Alexander	Single
Preaching choir	March 18, 1938	Lee Lord	Extended Play
No comment	March 5, 1930	Hunter Marsh	Single

1:25

...
...




Figure 5: Show more Album Page

Home
Search
Your library

Top Trending Track

Track Number	Album	Track Name	
1	Double trouble	Unused Home	+
1	No comment	Closer Earth	+
1	Preaching choir	Unused Home	+
1	Ice cold	Carebbian Saloon	+
1	Sleeping dogs	Cold Era	+
1	Blank canvas	Closer Earth	+
1	Concept art	Instrumental Heaven	+
1	No direction	Unused Home	+
2	Double trouble	The Hottest Soul	+
2	No comment	Closer Earth	+

1:25

...
...




Figure 6: Show more Track Page

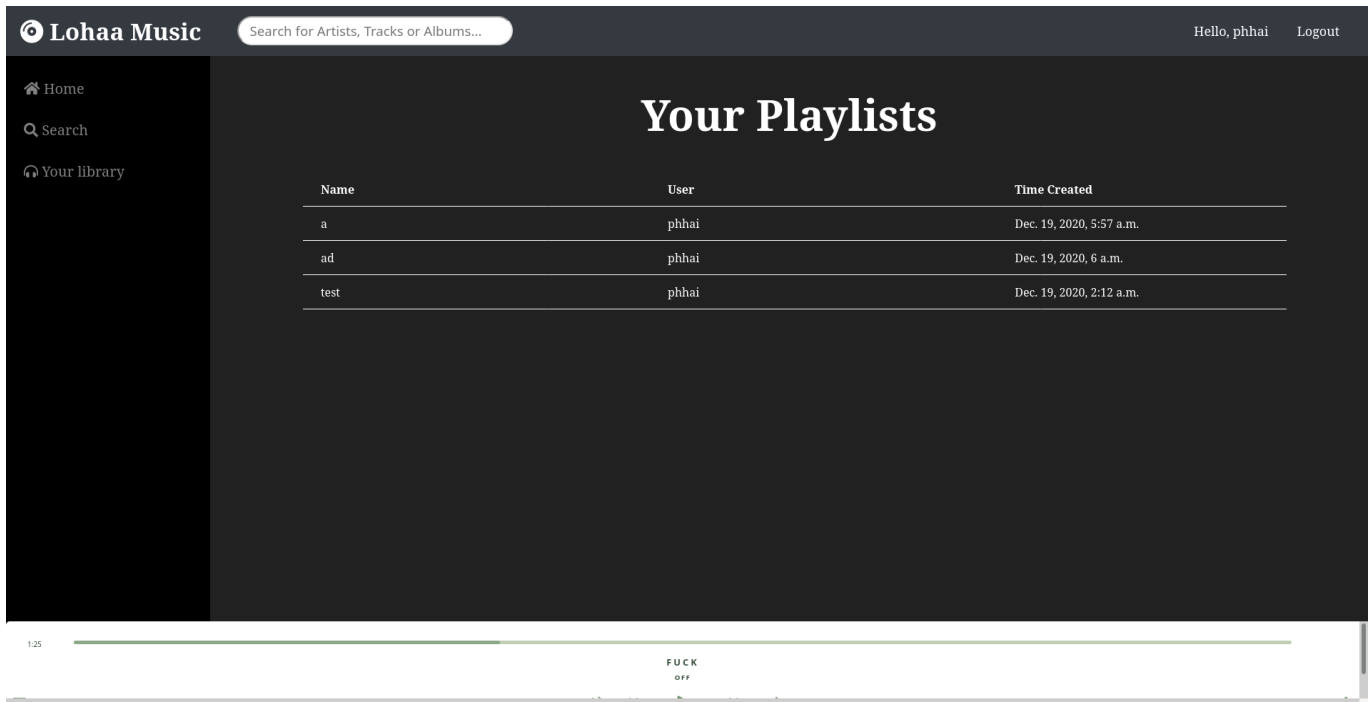


Figure 7: User's Playlist Page

Of course all the related entities such as track crediting an artist, album owned by an artist, playlist containing track,... will be resemble in the interface by links pointing to the detailed page of each object.

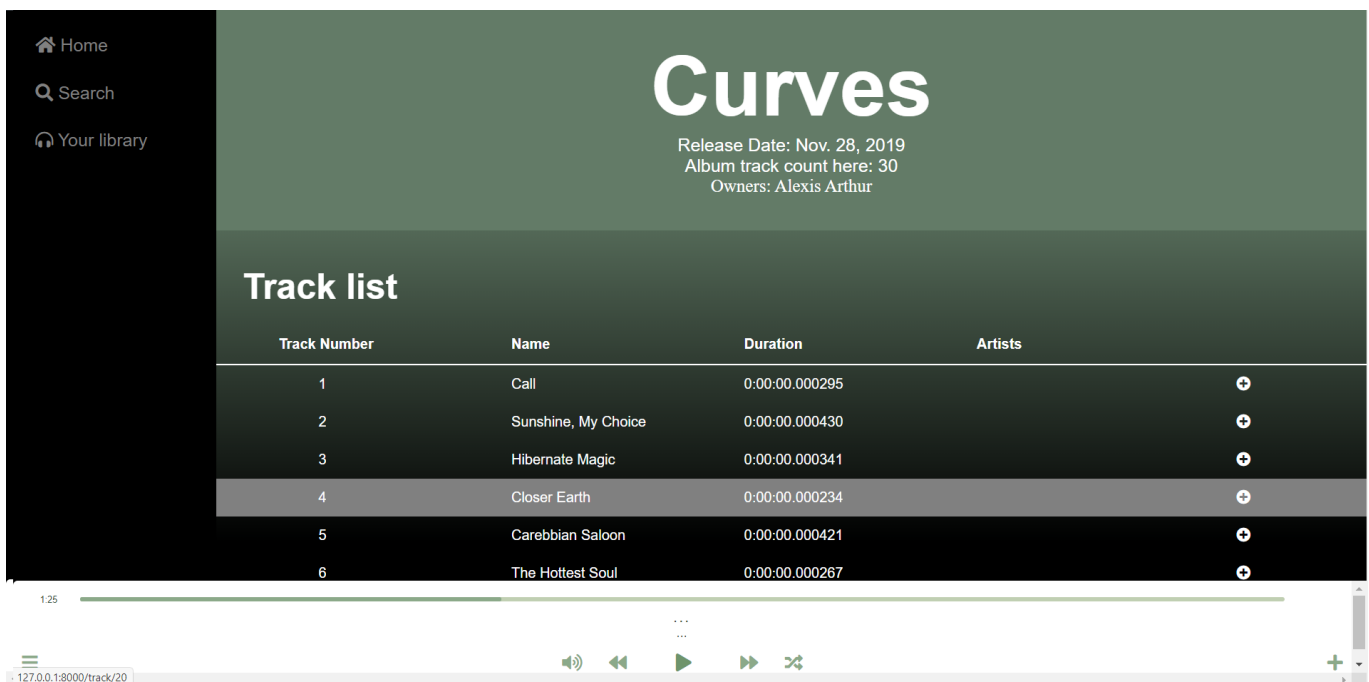


Figure 8: Album detail Page

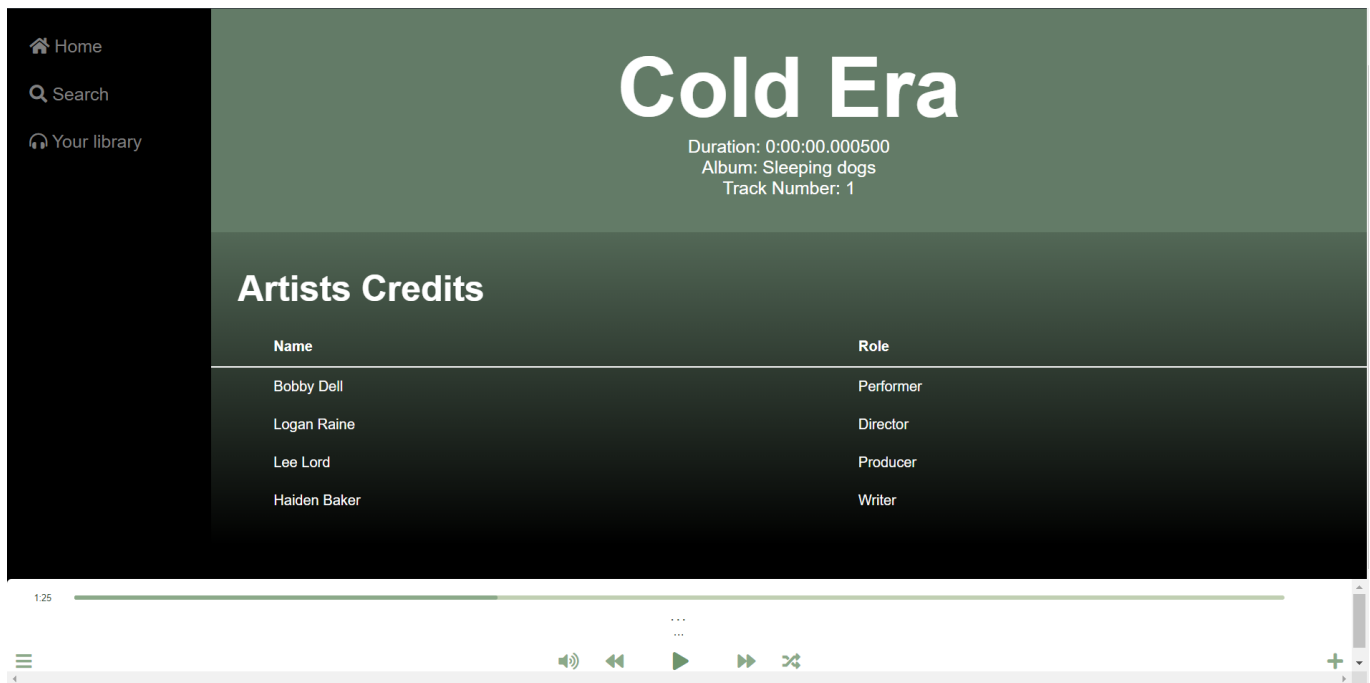


Figure 9: Track detail Page

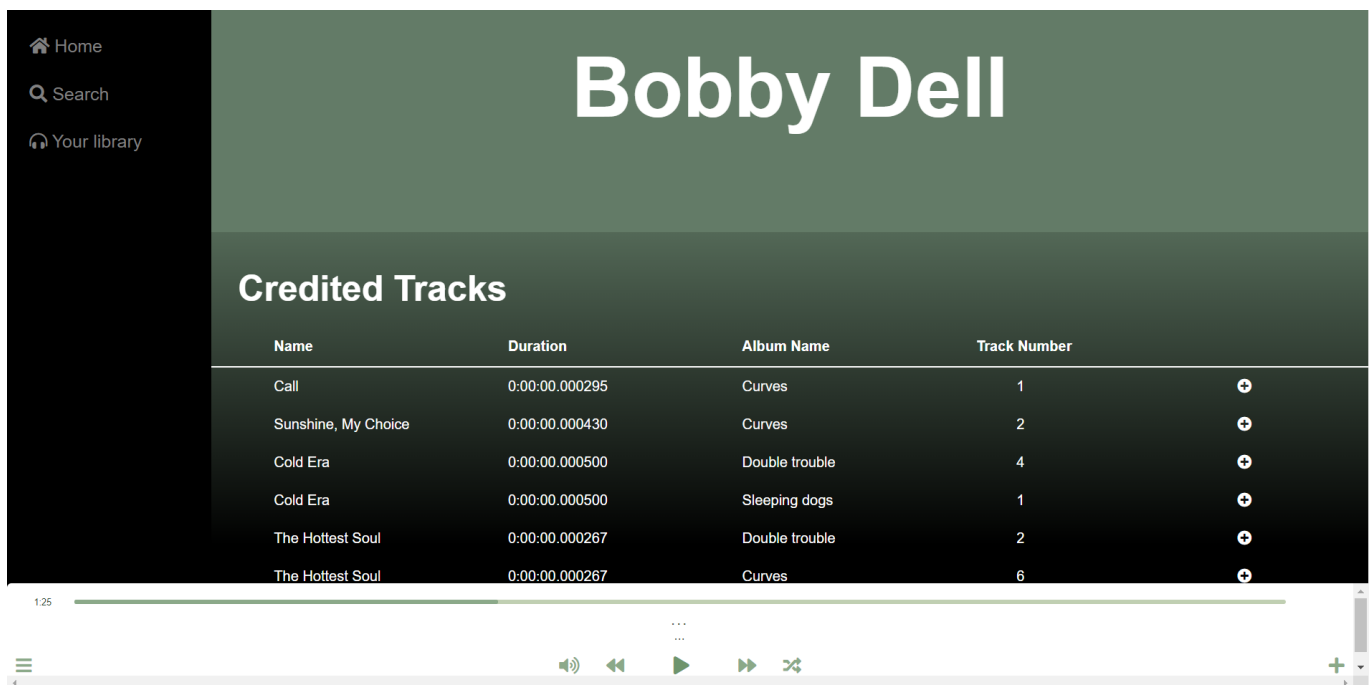


Figure 10: Artist detail Page

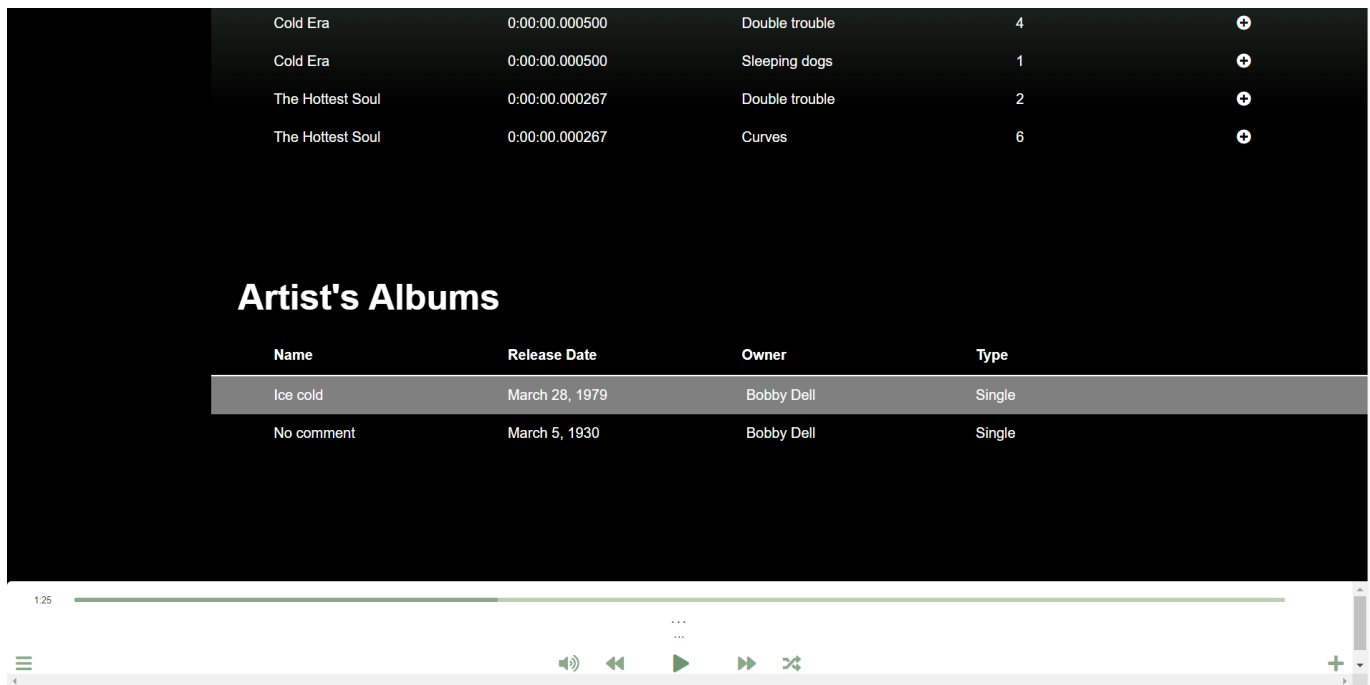


Figure 11: Artist detail Page

As for listening to tracks, our listener can choose 'add' to push the track to their listen queue.

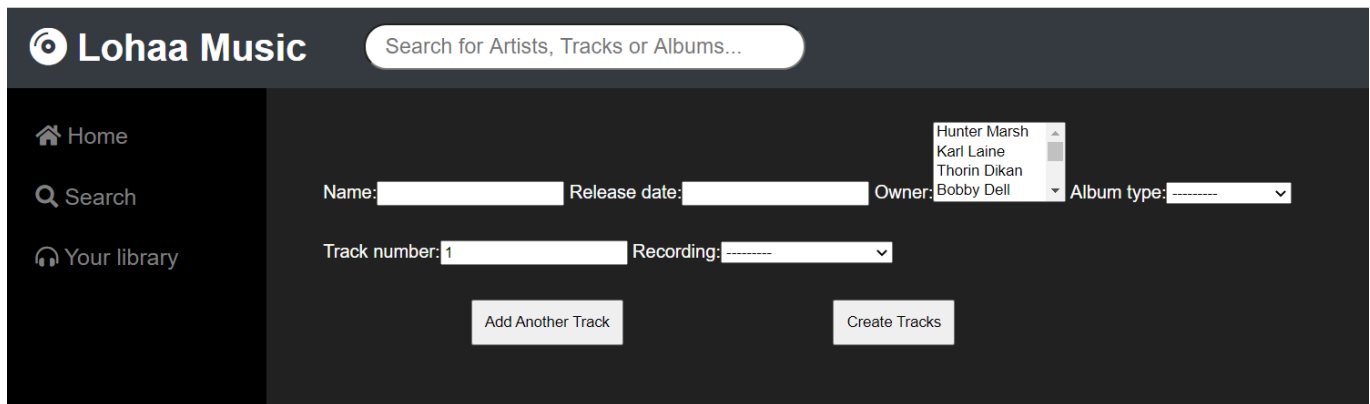


Figure 12: Add track to album

The user can also manage their queue: see queue, manually remove songs, choose next song to play,... The 'player-bar' also have many options to tweak such as pause, repeat, shuffle,...

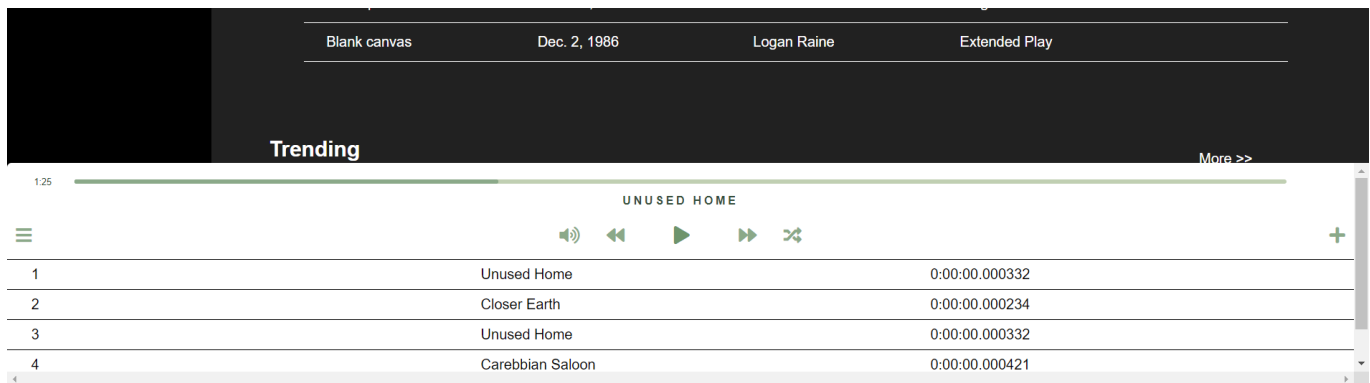


Figure 13: Listening Queue

6.2 Admin Interface

For the superuser of our system, after logging in, he/she have all the direct access to the database that we have. Some of that power includes:

- Add/Remove/Edit Any relation.
- Manage user and grant permission to users.
- Manage changes to DB and even rollback those changes.

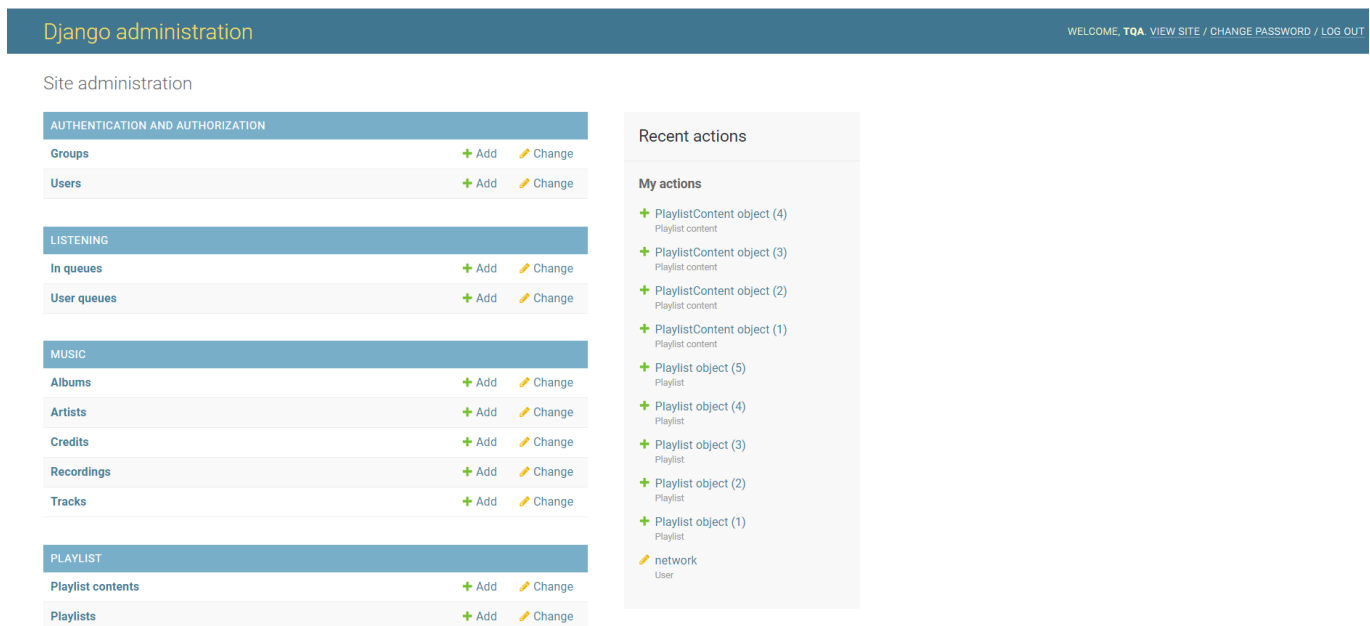
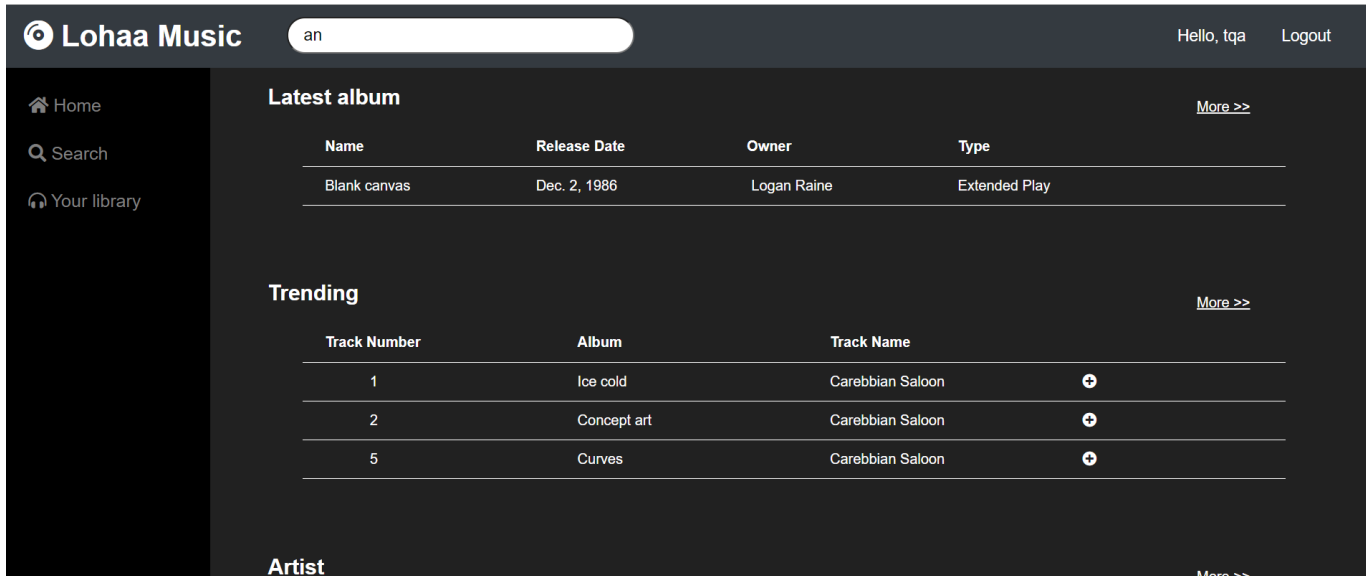


Figure 14: Admin Page

6.3 Searching Feature

For quick browsing in our service, we implemented a simple searching mechanism to allow user to search for all tracks/albums/artists/playlists at the same time by typing into the search bar.

This will show up the top matches result as well as the option to see all results of a certain type (all albums for example)



The screenshot shows the Lohaa Music application interface. At the top, there is a search bar with the text 'an' and a 'Hello, tqa' greeting with a 'Logout' link. The left sidebar contains navigation links: 'Home', 'Search', and 'Your library'. The main content area is divided into two sections: 'Latest album' and 'Trending'.

Latest album

Name	Release Date	Owner	Type
Blank canvas	Dec. 2, 1986	Logan Raine	Extended Play

[More >>](#)

Trending

Track Number	Album	Track Name	
1	Ice cold	Carebbian Saloon	+
2	Concept art	Carebbian Saloon	+
5	Curves	Carebbian Saloon	+

[More >>](#)

Artist

[More >>](#)

Figure 15: Search for 'an'

7 Security Improvements

Now that we have a functioning database, we can discuss about its securities.

7.1 Authorization and Role Assignment

Though our application is a free to use service, it still requires restrictions on who can access and change the system's database. Without authorization, all users have direct access to database and can potentially compromise the system's data integrity, as well as introducing ambiguity situations to our database. To combat this, we will apply the idea of RBAC (role-based access control) model to our system.

Authorization will be implemented by authorization info, including username, password and role. All users are required to log-in to their specific assigned role before being allowed to access database and modify data. A role here generally specializes what a user can do and what database relations they can modify in the process.

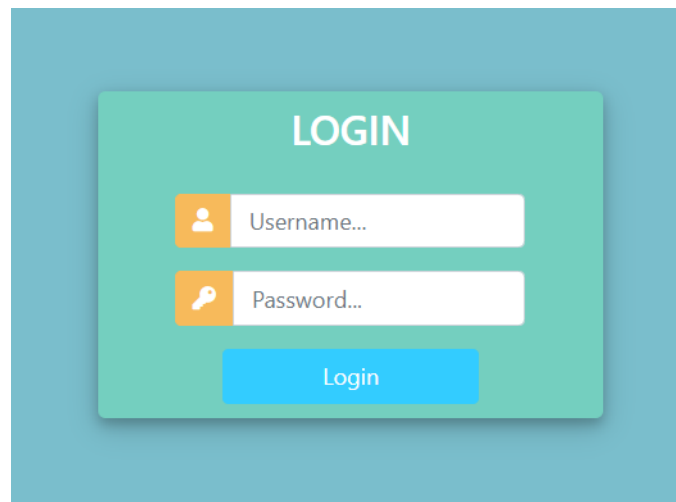


Figure 16: Login Page

In django, we implement user role by specifying them into different groups. The superuser (admin) himself is not a user within any group. Instead, he's specified using a Boolean value `is_admin` in Django's default user model

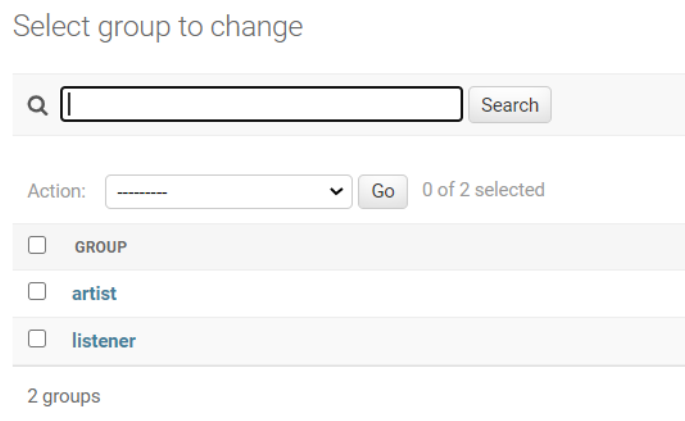


Figure 17: User Groups (Admin is not a group)

Our roles includes:

1. Admin (Super User): Has all permissions and direct access to the database, can change any data, add/remove user, grant permission and track database changes as well as rollback those changes.
2. Moderator: Receive DB change requests and accept (change database) or reject them.
3. Artist: Artist has all features of a listener as well as can manage their recording, tracks, albums.
4. Listener (General User): An average user can browse the page for track, album, playlist and artist. They can also add their own playlist and set them to public to share with the whole network.

7.2 SQL Injection

Another security problem is SQL injection where the input string query can be manipulated according to the SQL syntax to change our query's functionality and eventually leads to data breach and many more risks.

When taking input from user, it can contain SQL code that can be used to attack our database. To mitigate against this kind of attack, we will use DB-API's parameter substitution. It will sanitize the user input before execute the SQL statement.

8 Conclusion

In this assignment, we have created a database management system for a free music streaming service. We hope that we have included all the necessary knowledge in this report and meet the requirements. The database structure is quite simple, which makes it easy for also other programmers to understand it. First, we went through a process of constructing a database management system. Next, we wrote down all the functional and non-functional requirement description for the system and then moved to presented it using conceptual and logical design on draw.io tool. Finally, we implement it into SQLite by Python and got the results as above.

During our Database System course we have learned about the basics of database design. This project gave us the opportunity to try our new skills in practice. While doing this project we also gained deeper understanding on database design and how it can be implemented in real life situations. We believe we can also use our database design skill in other school projects.

9 Log

1. Updated conceptual design: added authentication
2. Updated logical design: splitted user relation into user and queue, added authentication
3. Updated requirements and specs to match new updates
4. added data requirements
5. Overhaul schema by adding recording entity and change the use of track entity.
6. Update so schema, logical design, added functional requirement graph and normalized
7. Added index and example queries as well as stored procedures and triggers
8. Application demo added
9. Added Reference

References

- [1] Ramez Elmasri; Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson, seventh edition, 2015.