UNIVERSITY OF ZAGREB
FACULTY OF ORGANIZATION AND INFORMATICS
VARAŽDIN

Marin Bogešić, Emilien Jacques, Elena Kržina, Lara Tičić

MULTIMEDIA STREAMING APPLICATION

PROJECT

THEORY OF DATABASES

Varaždin, 2023

UNIVERSITY OF ZAGREB

FACULTY OF ORGANIZATION AND INFORMATICS

V A R A Ž D I N

Marin Bogešić, Emilien Jacques, Elena Kržina, Lara Tičić

Student ID: 00161374414, XXX, 0016135585, 0016129266

Programme: Baze podataka i baze znanja, Riadenia a informatiky, Baze podataka i baze znanja, Informacijsko i programsko inženjerstvo

MULTIMEDIA STREAMING APPLICATION

PROJECT

Mentor:

prof. dr. sc. Markus Schatten, Milica S. Škembarević, Michal Kvet, PhD

Varaždin, January 2023

Marin Bogešić, Emilien Jacques, Elena Kržina, Lara Tičić

Statement of Authenticity

Hereby I state that this document, my Project, is authentic, authored by me, and that, for the purposes of writing it, I have not used any sources other than those stated in this thesis. Ethically adequate and acceptable methods and techniques were used while preparing and writing this thesis.

The author acknowledges the above by accepting the statement in FOI Radovi online system.

Abstract

Application for multimedia streaming made using PostgreSQL, Kafka, KsqlDB, Python, NestJS and Angular. PostgreSQL is used to save the stationary data such as User data, playlists, artists and more, while streaming is used to better show the data that changes in a fast pace such as likes, views and comments of certain audio tracks. The multimedia streaming app is made for anyone who enjoys listening to music. This work presents the main parts of implementation and creation of the database, the frontend and the backend, all integrated as one web application.

Keywords: kafka; streaming; ksqldb; streaming application; postgresql; kafka streaming application; multimedia

Table of Contents

1. Description of the Application Domain

In this chapter, we review the application domain of the application and what will be used to develop it.

1.1. The Domain

In this project, an application for streaming audio tracks will be developed. A database will be created using PostgreSQL, KsqlDB, and Kafka. NestJS and Angular will be used for the backend and frontend of the application respectively.

Before creating the database, it is necessary to define the domain. For the purposes of this project, an application will be created for streaming audio tracks, which will allow users to like and comment on individual songs. Users will be able to subscribe to other users so that they can easily follow the authors of the music they are interested in. Users will also be able to organize certain audio tracks into playlists. When creating a new user account, two playlists will be created automatically; Liked Music, in which all the songs liked by the user are added, and My Music, in which are the songs published by the user.

The target users of this application are all people who listen to music of any kind. The application will be easy to use and will offer a variety of content. Several technologies will be used in order to realize the application envisioned in this way.

1.2. The Technologies Used for Development

The technologies used for the app will be listed and introduced below.

PostgreSQL will be used to implement the static part of the database. PostgreSQL is an object-relational database management system, used for the primary data storage of many applications. We decided to use it in our project because it is open source, easy to use, and has all the qualities needed to create useful and functional applications - it is reliable, secure and has very good performance. Also, nowadays, it is very often used as for data storage in many different applications.[1]

We will also use **Apache Kafka**, which is an event streaming platform. It will be used for the active part of the database so as to show how the data is coming and going. As the application that will be created is intended for streaming audio records, Kafka will enable the storage, reading and analysis of data streams - a large amount of data that is generated at the same time from different sources and accessed by different users. The way Kafka stores and processes data ensures the ACID properties of databases (Atomicity, Consistency, Isolation, Durability).[2]

In order to connect Kafka and our database in PostgreSQL, **ksqlDB** will be used. KsqlDB is a system that enables the use of the SQL language to perform stream processing tasks. KsqlDB is built on Kafka and it makes it easier to implement data streaming applications

because it enables real-time data processing through the use of a simple SQL interface.[3]

Python is one of the most used and popular programming languages. It has a very wide application in various fields, and it is simple and object-oriented, with a large number of available libraries. In our project, Python will be used to mock the data needed for our database. Mock data is used to test the correctness of the software by using fake resources. Using Python, the tables are filled with random data in order to test the functionality of the application and to have a large enough number of tupples. [4]

Angular was used to develop the frontend of the application. Angular is a development platform built on TypeScript, and it consists of a framework for creating web pages, many different libraries and tools for developers. Today, Angular is one of the most widely used technologies for developing front-end applications. For this project, we chose angular because with the use of this framework, creating a page is quick and easy, and debugging is relatively simple.[5]

2. Theoretical introduction

This chapter focuses on presenting the theory of technologies used to implement the application.

In order to realize the previously described application, a hybrid (Apache Kafka and ksqldb) and object-relational (PostgreSQL) approach to databases will be used as we have established beforehand.

Relational databases are the standard for data storage and management. They consist of relations (tables) with unique names, in which columns represent attributes, and rows represent entity instances. In order for each instance, that is, each row, to be unique, primary keys are used. Those are unique identifiers that are assigned to each record in the table. Tables are interconnected using foreign keys, identifiers assigned to a particular row that reference a value in another table.[6]

The relational model is ideal for use if the data is well structured and tabular, but it is not practical for working with a large number of data or with highly connected databases. In these cases, working with the database, that is, executing queries, becomes very slow.[6]

PostgreSQL is an object-relational open source database management system. It brings together the best features of these database approaches; stores data in the form of relations and enables working with complex structures and data types.[1]

The PostgreSQL database system stores data in the form of tables. In order to create a table, it is necessary to assign it a unique name, and to define the names and data types of all its attributes. Below is an example of code from the official PostgreSQL site that creates a table that will store data about cities and their weather conditions. There is also an example of the INSERT command that populates that table with rows of data[1]. In this case, we will enter only one row.

Listing 1: Create table example

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
    prcp            real,          -- precipitation
    date            date
);

INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

[1]

In order to access the data from the tables, the SELECT command is used. The sign * represents all the data from the table (all), so if we want to retrieve all the data from the table created above, the following command is used.[1]

Listing 2: Select * example

```
SELECT * FROM weather;
```

[1]

The former statement would then return the entire tupple that we entered into the weather table.

In order to fetch only specific columns of data, we would only have to specifiy which columns we want in our result. Let's say, for example, that we want to fetch only the city, lowest temperature and highest temperature. To do that, we would use a query like the one below:

Listing 3: Select some columns example

```
SELECT city, temp_low, temp_hi FROM weather;
```

[1]

If we are only interested in specific data, the WHERE clause can be used to specify the conditions that the data must meet. The following example retrieves only those rows of data from the weather table where the attribute value of the city column is San Francisco, and the precipitation is greater than 0[1].

Listing 4: Select specific data example

```
SELECT * FROM weather WHERE city = 'San Francisco' AND prcp > 0.0;
```

[1]

Another very important and often used feature of PostgreSQL are triggers. With the help of triggers, the calling of certain functions can be automated, that is, it can be defined in which cases a certain function is automatically executed. In order to define a trigger, it is necessary to create a trigger function - a function that defines what must happen when the trigger is called, and then create a trigger - which defines when this function must be executed. The syntax to achieve this is shown in the code below.[1]

Listing 5: Trigger syntax

```
CREATE FUNCTION trigger_function()
   RETURNS TRIGGER
AS $$
BEGIN
   -- trigger logic
END;
$$ LANGUAGE PLPGSQL;

CREATE TRIGGER trigger_name
   {BEFORE | AFTER} { event }
   ON table_name
   [FOR [EACH] { ROW | STATEMENT }]
       EXECUTE PROCEDURE trigger_function
```

[1]

The example above demonstrates how to implement a function and trigger. In order for

the trigger to work, first it's necessary to create a function that returns a trigger. Then we define what exactly the function would have to do, and then define which language was used to write the function.

We create a trigger either before or after an event has occured, for each row or statement, on an event that could be an insert, an update or a delete. For instance, if we would like to check whether a value is already stored in the same table, we could use a trigger that would execute the function before an insert to the table. If we want to insert a tupple into a different table using some of the values we just added into the first table, we could use the after insert trigger.

In our project, these three main methods of programming in PostgreSQL will be used.

Hybrid databases are characterized by the fact that they combine the features of both in-memory and on-disc databases so that certain data can be accessed quickly, but also so that larger amounts of data can be stored. Hybrid databases are a combination of Relational and NoSQL databases. [7]

Apache Kafka is software that enables the storage of data, i.e. data streams, in real time, and thus can take over the function of a database. Kafka considers data as streams of records, and enables storage, processing and access to them.[2]

ksqlDB is a SQL interface for performing operations on Kafka data. It simplifies working with Kafka data. Its syntax is very similar to SQL, and in this way it enables programmers to work with data streams without acquiring a lot of new knowledge. This interface enables you to perform stream processing or data analysis operations on data stored in Kafka. [3]

The data stream in ksqlDB is a chronological sequence of events, that is, rows. Once a new row has been added to the stream, it can no longer be updated, but new rows can always be added to the end of the stream. Each row is associated with a key that identifies it. A row is stored in a partition, and all rows with the same key are stored in the same partition. [3]

The following example from the official ksqlDB page [3] shows the code that creates the data stream. There are three parameters in the CREATE STREAM command: kafka_topic contains the name of the Kafka topic that is the basis of the stream, value_format contains the type of encoding of messages that are stored in Kafka, and partitions, which contains the number of stream partitions.

Listing 6: Create stream example

```
1  CREATE STREAM riderLocations
2          (profileId VARCHAR, latitude DOUBLE, longitude DOUBLE)
3      WITH (kafka_topic='locations', value_format='json', partitions=1);
```

[3]

In order to better display the data from the stream created above, a table can be created. The SELECT statement selects the data of only the last, most recently added row and adds it to the table, which is shown in the code below. To access only the last line of the stream, the function LATEST_BY_OFFSET is used.

Listing 7: Create table out of stream example

```
CREATE TABLE currentLocation AS
SELECT profileId,
         LATEST_BY_OFFSET(latitude) AS la,
         LATEST_BY_OFFSET(longitude) AS lo
  FROM riderlocations
  GROUP BY profileId
  EMIT CHANGES;
```

In addition to database management systems and streaming platforms, Python will be used to generate data, and Angular and NestJS will be used to implement the application.

3. Database model

In this chapter, we will discuss the entity-relationship model for the project and will then show each table and its contents and types separately.

3.1. ER Model

For our project's database where we implemented our entities, relations, functions and triggers; we used PostgreSQL. Psql is an open source relational database. Despite it being relational itself, it supports both SQL (relational) and JSON (non-relational) querying. In our project we used PostgreSQL for creating our crucial entities as well as their triggers. The list of main entities is following:

- Users
- Roles
- Comments
- Playlists
- Music
- Genres
- Artists

The entities that are used to connect other main entities with a more to more relation are called weak entities. Our weak entities are following:

- Subscriptions
- PlaylistMusic
- MusicArtists
- MusicGenre

Considering relations between entitites: a weak entity Subscriptions allows for one user to subsrcibe to more users, similar to how in an entity MusicArtists: one Artist can make multiple music entities, while one music entity can be made by multiple Artists. Weak entities Playlist-Music and MusicGenre work in a similar way: one playlist can own multiple music entities as music entities can be on multiple playlists. Also A music entity can be of multiple types of Genre and one Genre can have more music entities. As an image above mentions, Artist entity is an entity and not just an attribute solely because of a functionality that we can search music by an Artist.

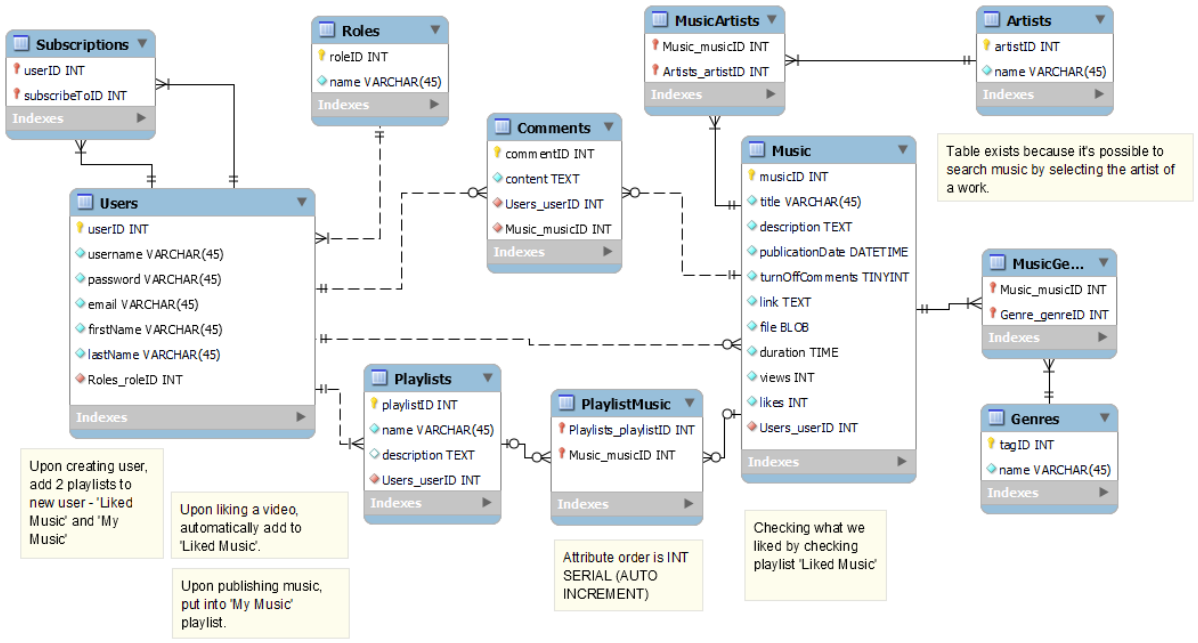

Figure 1: ER Diagram; own work

3.2. Tables used for the database

In this section, we will touch on the description and code implementation of some of the tables that were mentioned in the previous section.

3.2.1. Table Description

The Users table has one primary key and one foreign key (as described previously). The relationship with the Roles entity is 1:m. One user can have one role, but one role can be assigned to more users. It's used to distinguish the actions an administrator and registered user can do.

The Roles table has only one primary key that is an integer. It's a simple table that doesn't have any foreign keys.

The Playlists table has one primary key and one foreign key. The foreign key is used to connect with Users table with 1:m relation. A user can have more playlists.

Music table has many attributes. It has one primary key as usual, and one foreign key connected through a weak entity called PlaylistMusic. Because of the weak entity, it achieved a m:n relation. One Music entity can be interacted by more users, and one user can interact with more Music entities. It's important to note that the Music table will be implemented with ksql and PostgreSQL, so PostgreSQL will not have every attribute in the table as shown below.

With a similar relation as previously mentioned, table Genres is connected with table Music through a weak entity called MusicGenres. With it they achieve an m:n relationship. A music entity can own more genres, and one genre can have more music entities.

Comments are also used as a weak entity, but they themselves have an attribute apart from the primary key (content). They are used to connect Users table and Music table. So that one user can interact with more music entities and one music entity can be interacted by more users. With the table comments there is also a weak attribute PlaylistMusic that was previously mentioned. Both tables are crucial in connecting the User table and its entities with Music table and their entities.

Table Artists has one primary key and an attribute 'name'. As previously mentioned, there is a table Artists instead of an attribute inside Music table because there is a functionality that allows users to search music by an Artist.

3.2.2. Table Review

Table 1: Users

Value 1	**Value 2**	**Value 3**
α	β	γ
userID	int	PK
username	varchar	
password	varchar	–
email	varchar	–
firstName	varchar	–
lastName	varchar	–
Roles role	int	FK

Table 2: Roles

Value 1	**Value 2**	**Value 3**
α	β	γ
roleID	int	PK
name	varchar	

Table 3: Playlists

Value 1	**Value 2**	**Value 3**
α	β	γ
playlistID	$\int$	PK
name	varchar	
description	varchar	
Users user	$\int$	FK

Table 4: Music

Value 1	**Value 2**	**Value 3**
α	β	γ
musicID	int	PK
title	varchar	
description	text	
publicationDate	datetime	
turnOffComments	tinyint	
link	text	
file	blob	
duration	time	
views	int	
likes	int	
Users userID	int	FK

Table 5: Genres

Value 1	**Value 2**	**Value 3**
α	β	γ
tagID	int	PK
name	varchar	

Table 6: Comments

Value 1	**Value 2**	**Value 3**
α	β	γ
commentID	int	PK
content	text	
Users userID	int	FK
Music musicID	int	FK

Table 7: Artists

Value 1	**Value 2**	**Value 3**
α	β	γ
artistID	int	PK
name	varchar	

4. Implementation

Basic parts of database implementation and corresponding applications will be shown in this part of the documentation.

4.1. Database Creation

The database has two parts: the static part, which is used for data that won't change frequently, and the dynamic part implemented using Kafka and Ksqldb for rapidly changing data or data streaming.

4.1.1. The Static Part - PostgreSQL

As previously mentioned, to create a table we are using the command 'CREATE TABLE'. Then we declare the attributes inside the table such as playlist id. There are many different types of attributes; an id attribute commonly has an integer type. It is also useful to put NOT NULL extension command, as that excludes the possibility of this attributes absence. It is also useful and in our case crucial to increment the values; we won't have to always add a new index ourselves.

Tables were creating the following way:

Listing 8: Roles Table

```
1  CREATE TABLE Roles(
2          roleId SERIAL PRIMARY KEY,
3          name VARCHAR(15)
4  );
```

The Roles table is very simple. It has a roleId which is the primary key, and a varchar typed name.

Listing 9: Users Table

```
1  CREATE TABLE Users(
2          userId SERIAL PRIMARY KEY,
3          username VARCHAR(45) not null,
4          password VARCHAR(64) not null,
5          email VARCHAR(45) not null UNIQUE,
6          firstName VARCHAR(30) not null,
7          lastName VARCHAR(30) not null,
8          roleId INT REFERENCES Roles(roleId)
9      ON UPDATE CASCADE ON DELETE CASCADE
10 );
```

The Users table has an ID for the user, a username, password and email which has to be unique to the user, then a first name, a last name, and a foreign key roleId which references roles. Upon updating or deleting the key, the value of the key will cascade, which will simultane-

ously delete or update an entry from both the child and the parent table. The child table would be Users, while the parent table would be Roles.

Listing 10: Subscriptions Table

```
CREATE TABLE Subscriptions(
        userId INT REFERENCES Users(userId) ON UPDATE CASCADE ON DELETE CASCADE,
        subscribedToId INT REFERENCES Users(userId) ON UPDATE CASCADE ON DELETE
         CASCADE,
        PRIMARY KEY(userId, subscribedToId),
        CHECK (userId <> subscribedToId)
);
```

Subscriptions are a weak table which connects one User to the other. It has two columns, both are the primary key, and there is a check condition which negates the possibility of a user subscribing to themselves.

Listing 11: Artists Table

```
    CREATE TABLE Artists(
        artistId SERIAL PRIMARY KEY,
        name VARCHAR(45)
);
```

Listing 12: Genre Table

```


CREATE TABLE Genres(
        genreId SERIAL PRIMARY KEY,
        name VARCHAR(45) not null
);
```

Listing 13: Music Table

```

CREATE TABLE Music (
        musicId SERIAL PRIMARY KEY,
        title VARCHAR(45) not null,
        description TEXT,
        publicationDate TIMESTAMP not null,
        turnOffComments BOOLEAN DEFAULT FALSE,
        link TEXT not null,
        duration TIME not null,
        userId INT REFERENCES Users(userId)
 ON UPDATE CASCADE ON DELETE CASCADE
);
```

The Music table is a very important table which is connected to many tables. It has an ID, the title of the music, description of the audio, its publication date and the option of turning off comments which is of boolean type, a link, the duration which cannot be null and a userId which is the foreign key to the Users table and tells us who uploaded the music.

Creating other tables is very similar. For inserting the data, INSERT statements are used.

Listing 14: Insertion Example

```
1
2 INSERT INTO Users (username, password, email, firstName, lastName, roleId) VALUES ('
   XHeyX', '28c3c24bf3e3a40f9628a8f3e62f116c3164beb899fbcfe9cc8dd15d905e03a5',
3 'allyalles@gmail.com', 'Ally', 'Alles', 2); INSERT INTO Artists (name) VALUES ('
   Years and Years'), ('Queen'), ('Britney Spears'), ('Gojira'), ('John Newman'), ('Ed
    Sheeran'), ('Mozzart'), ('Elton John'), ('Micheal Jackson'), ('The Weeknd'), ('
   Kungs'), ('Kygo'), ('by User');
```

The created database also has multiple functions and connected triggers. The following functions are implemented inside our project and documented.

The following is an example of a simple trigger that searches the entity Playlists with the conditions that it is the right user ID and the name of the playlist is 'My Music'.

Listing 15: Check if in list function and connected trigger

```
1  CREATE FUNCTION public.checkifinlist() RETURNS trigger
2      LANGUAGE plpgsql
3      AS $$
4          DECLARE postoji BOOLEAN;
5  BEGIN
6          postoji := (SELECT musicId FROM PlaylistMusic WHERE musicId = NEW.musicId
            AND NEW.playlistId = playlistId);
7          IF NOT postoji THEN
8                  INSERT INTO PlaylistMusic (playlistId, musicId, ordered) VALUES (NEW
                    .playlistId, NEW.musicId, NEW.ordered);
9          END IF;
10         RETURN NEW;
11 END;
12 $$;
13
14 CREATE TRIGGER insertion BEFORE INSERT ON public.playlistmusic FOR EACH ROW EXECUTE
    FUNCTION public.checkifinlist();
```

The trigger is checking with a boolean whether or not a certain music entity exists in the playlist. If the boolean value, selected by checking whether the identification numbers of the newly inserted music is the same as one that is already in the playlist, is false, that is if the music entity is not found in the playlist, then the function is inserting new values into the playlist. It prevents iserting a double value of the track into a certain playlist.

The trigger specifies the checking of the function before the insert into the playlist.

Another trigger and function are shown below.

Listing 16: Making playlists function and connected trigger

```
1  CREATE FUNCTION public.makeplaylists() RETURNS trigger
2      LANGUAGE plpgsql
3      AS $$
4  DECLARE x INT := (SELECT userId FROM Users WHERE email = NEW.email);
5  BEGIN
6          INSERT INTO Playlists (name, description, userId)
7                  VALUES ('Liked Music', 'Music liked by you', x),
```

```
8                           ('My Music', 'Music you have uploaded', x);
9           RETURN NEW;
10  END;
11  $$;
12
13  CREATE TRIGGER creatinguser AFTER INSERT ON public.users FOR EACH ROW EXECUTE
     FUNCTION public.makeplaylists();
```

The above example shows that on creation, two playlists are instantly initiated: 'Liked Music' and 'My Music'. Upon creating a new user, those playlists are immediately added into the database. So the trigger specifies that after creating a user, we make two new playlists; meaning we call the above function.

And finally, a trigger for adding music to 'My Playlist' is created.

Listing 17: Inserting into My Music function and connected trigger

```
1   CREATE FUNCTION public.addmusictomyplaylist() RETURNS trigger
2       LANGUAGE plpgsql
3       AS $$
4           DECLARE playlista INT;
5   BEGIN
6           playlista := (SELECT playlistId FROM Playlists
7                                   WHERE new.userId = userId
8                                   AND name = 'My Music');
9           INSERT INTO PlaylistMusic (playlistId, musicId)
10                  VALUES (playlista, NEW.musicId);
11          RETURN NEW;
12  END;
13  $$;
14
15  CREATE TRIGGER addingmusic AFTER INSERT ON public.music FOR EACH ROW EXECUTE
     FUNCTION public.addmusictomyplaylist();
```

In order to quicken the process of adding a specific new track into the 'My Music' without the need to write code in the back end, we added a trigger that would automatically put a user's uploaded audio file into the playlist 'My Music'. After we insert the audio file, we automatically execute the function for each added row.

4.1.2. The Dynamic Part - Kafka streaming

As mentioned, Ksql allows us to use Kafka topics and turn them into SQL tables. With ksql we are fetching and processing data present inside certain topics. Below are snippets of code used for the implementation to explain what we did.

Listing 18: ksql streams and tables

```
1
2   CREATE TABLE MusicDesc (
3           musicId INT PRIMARY KEY,
4           name VARCHAR,
5           description VARCHAR,
```

```
6           publicationDate VARCHAR,
7           link TEXT,
8           turnOffComments BOOL,
9           duration VARCHAR
10          ) WITH (
11          kafka_topic='music',
12          value_format = 'json',
13          partitions = 1
14  ) EMIT CHANGES;
15
16  CREATE STREAM MusicStats(
17          musidId INT,
18          views BIGINT,
19          likes BIGINT
20          ) WITH (
21          kafka_topic='musicStats',
22          value_format = 'json',
23          partitions = 1
24  );
25
26  CREATE STREAM MUSIC AS SELECT musicId, name, description, link, turnOffComments,
     duration, views, likes FROM MusicStats INNER JOIN MusicDesc ON musidId = musicId;
```

The following code demonstrates how to make a stream and how to make a table in ksql. We created a table named MusicDesc that had all the parts of the music description. All but likes and views of type bigint, which will be streamed in the MusicStats stream. To create a complete stream then, we combine everything by joining up MusicDesc and MusicStats by their primary key and integer, since streams can't have primary keys, and the stream is done.

Streams are created with kafka topics, which are something like containers for the streams, the format of the file will be a JSON file, and the stream will use one partition.

Listing 19: Statistics and inserts

```
1   INSERT INTO Music VALUES (1, 'My music', 'mine music', 'htpps://www.link.fr', false,
      '02:22', 3, 5);
2
3   CREATE TABLE Statistics AS
4   SELECT
5           musicId,
6           latest_by_offset(name) as musicName,
7           latest_by_offset(views) as musicViews,
8           latest_by_offset(likes) as musicLikes
9   FROM Music
10  group by musicId
11  EMIT CHANGES;
12
13  CREATE TABLE LikesMusic AS
14  SELECT
15          musicId,
16          latest_by_offset(likes) as musicLikes
17  FROM Music
18  group by musicId
```

```
EMIT CHANGES;

CREATE TABLE ViewsMusic AS
SELECT
        musicId,
        latest_by_offset(views) as musicViews
FROM Music
group by musicId
EMIT CHANGES;
```

This is code that creates a static table Statistics for showing the statistics of music made my selecting from the music table, where we select the music's ID, its name, views and likes and group them by the music ID since tables must be created with an aggregate function. We emit changes to see what's happening to the base.

The example also shows liked music and views music in case we want to look at the streamed data separately.

Before anything, it inserts one value into the music table.

Listing 20: Push queries

```

SELECT * FROM Statistics where musicLikes > 0 and musicViews > 0 EMIT CHANGES;
SELECT * FROM LikesMusic where musicLikes > 0 EMIT CHANGES;
SELECT * FROM ViewsMusic where musicLikes > 0 EMIT CHANGES;
```

Push queries to the following tables are made with the EMIT CHANGES keywords at the end. The following queries select everything from the three tables we've implemented above where the likes and views count is larger than 0. The queries give results continously, and the client recieves a message for every change in the database.

Listing 21: Comments

```
CREATE STREAM CommentStream(
        commentId INT,
        content VARCHAR
        ) WITH (
        kafka_topic='commentStream',
        value_format='json',
        partitions = 1
);
```

In the above example, we also create a music stream that has a simple comment ID integer and content which is of type varchar. It's created with the topic commentStream, its format is JSON and uses one partition.

```
CREATE TABLE Comment AS
SELECT commentId,
LATEST_BY_OFFSET(content) AS recentComment
FROM CommentStream
GROUP BY commentId
EMIT CHANGES;
```

```
7
8  SELECT * FROM Comment WHERE recentComment IS NOT NULL EMIT CHANGES;
9
10 INSERT INTO CommentStream VALUES (1, 'Very cool!');
11 INSERT INTO CommentStream VALUES (2, 'This is exciting...');
12 INSERT INTO CommentStream VALUES (3, 'They never disappoint.');
13 INSERT INTO CommentStream VALUES (4, null);
14 INSERT INTO CommentStream VALUES (5, 'I've been meaning to ask - how??');
15 INSERT INTO CommentStream VALUES (7, 'Nice!');
```

In the code above, we've created a table Comment from the stream CommentStream. We can also select everything from the comments where recentComment isn't null (those values will be left out), and to see whether it works, we insert the following seven values into the CommentStream. The ksqldb where we ran the push query should now return all those comments from above except comment with the ID of 4, which has a null value and does not fit the criteria of retrieving data.

Ksql data is connected to the backend with Confluent.

4.2. Mocking Data

To mock data, we've used the Python programming language. The main program looks like this:

```
1  from generate_data import *
2  from model.model import BaseModel
3  from typing import Sequence
4
5  def get_ids(m_list: Sequence[BaseModel]):
6      return list(map(lambda o: o.id(), m_list))
7
8  def sql_insert_into(*args: Sequence[BaseModel]):
9      for m_ls in args:
10         for m in m_ls:
11             print(m.insert_sql_sentence())
12
13 roles = generate_roles(["admin", "registered", "moderator"])
14 users = generate_users(10, get_ids(roles))
15 subs = generate_subscriptions(6, get_ids(users))
16
17 musics = generate_music(20, get_ids(users))
18 comments = generate_comments(30, get_ids(users), get_ids(musics))
19 artists = generate_artists(9)
20 genres = generate_genres(5)
21 playlists = generate_playlists(4, get_ids(users))
22
23 music_artists = generate_music_artists(9, get_ids(musics), get_ids(artists))
24 music_genres = generate_music_genre(10, get_ids(musics), get_ids(genres))
25
26 sql_insert_into(roles, users, subs, musics,
27                 comments, artists, genres, playlists, music_artists, music_genres)
```

The script imports from generatedata, model.model, which is a folder we use to store .py scripts in, and typing. Then we define two functions, get IDs which returns a list that maps IDs and sql insert into which then prints our insert sql sequence generated in the program.

Tables take data from other scripts that were created and generate for each table separately, which is then used in the sql insert into function to print out the generated values.

For the users, we prepared a .py file that would generate a user with a name and surname with a list, randomly picking from the list.

Listing 22: Names and surnames list

```
firstnames = ["Ozzie",
              "Alyce",
              "Binky",
              "Tabina",
              "Bentley"...]
lastnames = ["Simmins",
             "Stelljes",
             "Carrodus",
             "Waud",
             "Payler",
             "Fyndon"...]
```

The following code is an example of .py files for the data. This specific data is about comments.

Listing 23: Comments .py file

```
from model.model import BaseModel

class Comments(BaseModel):
    commentID: int
    content: str
    Users_userID: int
    Music_musicID: int

    def __init__(self,
                 commentID: int,
                 content: str,
                 Users_userID: int,
                 Music_musicID: int):

        self.commentID = commentID
        self.content = content
        self.Users_userID = Users_userID
        self.Music_musicID = Music_musicID

    def id(self) -> int: return self.commentID
```

Firstly, the .py script imports the base model of the mock data application. A class is created which takes the base model as an argument and defines commentId, content, user ID and music ID.

When initialized, the Comments class makes its attributes the given arguments, and then the class returns itself.

This is how comment data is generated:

Listing 24: Generating

```
def generate_comments(N: int, usersid: List[int], musicsid: List[int]) -> List[m.
 Comments]:
    id = [0]

    def next_id(ref: List[int]) -> Callable[[], int]:
        def f() -> int:
            ref[0] += 1
            return ref[0]
        return f
```

Listing 25: Randomizing string

```
def rdmstr() -> str:
        return "".join([choice("azertqsdfghklmwxcvbn")
        for _ in range(randint(5, 50))])
```

Listing 26: Specifications

```
    spec = SpecificationDict({
        "commentID": nextid(id),
        "content": rdmstr,
        "UsersuserID": choicewrapper(users_id),
        "MusicmusicID": choicewrapper(musics_id)
    })

    return generatedata(N, spec, m.Comments)
```

We create a function to generate comments that returns a list of comments, with the ID being 0 in a list in the beginning. We define a function that increments that ID. Then we define the function random string which joins a choice between a string of characters in range of 5 to 50. In the end, we have a specification dictionary, which uses the incrementing ID function, random string for content, and userID and musicID choicewrapping for both IDs respectively.

4.3. Creating the Application - Front End, Back End, Connections

The application was created using Angular for front end and NestJS for back end. The audio files that were used for the project were the mp3 format because the files were easy to splice thanks to the frame it contains and because each frame has a header. To connect PostgreSQL and Kafka and KsqlDb, the proper connectors for each were used. The static part of the database implemented with PostgreSQL was connected to the application via TypeORM, while the streaming part of the database used Confluent.io for connecting to the back end.

4.3.1. NestJS implementation

A large part of the project rests on the implementation of the application in NestJS. The rest of this section will be primarily focused on the NestJS code.

As an example, the following code snippet shows the implementation of a header and controller.

Listing 27: Users Controller

```
import {
  Controller,
  Get,
  Post,
  Body,
  Patch,
  Param,
  Delete,
} from '@nestjs/common';

import { UsersService } from './users.service';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';
import { UsersDto } from './dto/user.dto';

@Controller('api/users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Post()
  create(@Body() createUserDto: CreateUserDto) {
    return this.usersService.create(createUserDto);
  }

  @Get()
  findAll() {
    return this.usersService.findAll();
  }

  @Get(':id')
  async findOne(@Param('id') id: string): Promise<UsersDto | null> {
    return await this.usersService.findOne(+id);
  }

  @Patch(':id')
  update(@Param('id') id: string, @Body() updateUserDto: UpdateUserDto) {
    return this.usersService.update(+id, updateUserDto);
  }

  @Delete(':id')
  remove(@Param('id') id: string) {
    return this.usersService.remove(+id);
  }
```

```
}
```

The above example creates a controller for the Users table. First, we import libraries from the nestjs/common, and then we import from our own files.

We define the controller for the users by exporting the class UsersController and making a constructor that will be private and read-only. On POST, it returns the current object this from usersService and creates the createUserDto.

On GET it finds and returns all values from the object. If the GET has a parameter that we named 'id', then the controller will return the one value from the object that matches the id. It also works with wait, which NestJS uses so that requests can't be accepted until the connection with the database has been established.

Patch will get a parameter we named id and update the values where the id of the user matches the id from our parameter. Delete will drop the users with the id matching the id parameter.

The following is the code for the Users Service. Because of its relative complexity, it will be divided into smaller chunks and then explained separately.

Listing 28: Users Serice Creation

```
const MY_MUSIC_NAME = 'My Music';
const LIKES_NAME = 'Liked Music';

@Injectable()
export class UsersService {
  constructor(
    @InjectRepository(Users) private usersRepository: Repository<Users>,
    @InjectRepository(Roles) private rolesRepository: Repository<Roles>,
  ) {}

  async create(createUserDto: CreateUserDto): Promise<UsersDto> {
    const basicRole = await this.rolesRepository.findOne({
      where: { roleid: 1 },
    });
```

The above code firstly initializes two constant variables, My-Music-Name and Likes-Name which take the names of the playlists. Then we create an Injectable which attaches metadata, exporting the class UsersService that has a constructor which takes two repositories: Users and Roles.

Then the code asynchronously creates a promise, used for executing a series of asynchronous tasks in sequential order, and a new constant basicRole is created and is initialized as the rolesRepository, where we fetch the row that has 1 as the value for the role id.

Listing 29: Users Service Creating User Repository

```
    const user = this.usersRepository.create({
      ...createUserDto,
      role: basicRole,
    });
```

```
    await this.usersRepository.save(user);

    return toUserDto(user);
  }

  findAll() {
    return `This action returns all users`;
  }

  async findOneEntityByEmail(email: string): Promise<Users> {
    return this.usersRepository.findOne({ where: { email } });
  }
```

This snippet of code represent the creation of the user. It creates the usersRepository by creating the UserDto with the role of the basic role. It saves the current user and returns toUserDto, which defines how user objects are sent over the network. The DTO for users looks like this:

```
export interface UsersDto {
  userID: number;
  username: string;
  email: string;
  firstName: string;
  lastName: string;
  Roles_roleID: number;
}
```

FindAll() returns all users, and is left out of the code for simplification. FindOneEntityByEmail() returns the user which has an email that is the same as the string we gave the function as a parameter.

Listing 30: Users Service Working with Playlists

```
  async findUserMyMusicPlaylist(userid: number): Promise<PlaylistDto | null> {
    const user = await this.usersRepository.findOne({
      where: { userid },
      relations: { playlists: { user: true } },
    });
    if (!user) return null;

    const playlists = user.playlists;
    if (playlists.filter((a) => a.name === MY_MUSIC_NAME).length === 0)
      return null;

    return toPlaylistDto(playlists.filter((a) => a.name === MY_MUSIC_NAME)[0]);
  }

  async findUserLikesPlaylist(userid: number): Promise<PlaylistDto | null> {
    const user = await this.usersRepository.findOne({
      where: { userid },
      relations: { playlists: { user: true } },
    });
    if (!user) return null;
```

```
21
22      const playlists = user.playlists;
23      if (playlists.filter((a) => a.name === LIKES_NAME).length === 0)
24        return null;
25
26      return toPlaylistDto(playlists.filter((a) => a.name === LIKES_NAME)[0]);
27    }
```

The playlist functions findUserMyMusicPlaylist takes in an argument of numerical type and defines a constant user which takes the id of the user repository that matches the forwarded id and where the relation between the user and the playlist is true. If there is no user that matches the description, the function returns null and ends.

The playlists constant is initialized as the playlists that are bound to the specific user. All users have a playlist named 'My Music'. If its length is 0, or if there are no values in the playlist music, return null. Otherwise, return the toPlaylistDto and its first value.

The likes playlist functions very similarly and will thus not be explained in detail.

Listing 31: Users Service, Update, Delete and userEntity

```
1     async findOne(id: number): Promise<UsersDto | null> {
2       const userEntity = await this.usersRepository.findOne({
3         where: { userid: id },
4         relations: {
5           role: true,
6         },
7       });
8
9       if (!userEntity) return null;
10
11      return toUserDto(userEntity);
12    }
13
14    update(id: number, updateUserDto: UpdateUserDto) {
15      return `This action updates a #${id} user`;
16    }
17
18    remove(id: number) {
19      return `This action removes a #${id} user`;
20    }
21  }
```

Lastly, we define an asynchronous function which returns a promise based on the id number parameter that we forward it. userEntity is defined as fetching the user that matches the id we forwarded to the function and where the users have a role id assigned to them. Then, if the user entity is null, return null, and if not, the function returns the toUserDto of the entity.

Updating is done by taking in a number parameter that will represent an id and the updateUserDto. It returns the value 'This action updates a ID user.' Remove only needs an id to remove the user that has that specific Id.

The next example shows the implementation of the user end module. All the mod-

ule does is set the controllers which the module will use (in our case, that is the UsersController), the providers for the module (which is the UsersService), and the exports (which is the UsersService). Then the class is exported. It connects the other two parts of the code that were listed above. CLI automatically generates classes which are then overwritten for implementation of functions.

Listing 32: User Module

```
@Module({
  imports: [
    TypeOrmModule.forFeature([
      Users,
      Comments,
      Playlists,
      Subscriptions,
      Roles,
    ]),
  ],
  controllers: [UsersController],
  providers: [UsersService],
  exports: [UsersService],
})
export class UsersModule {}
```

The connectors and handlers are made for each table in the database.

5. Screenshots of the Application

In this section we show screenshots of the application. It shows the application's functions, its visual design and some of the possible tasks one can do and see in the program.

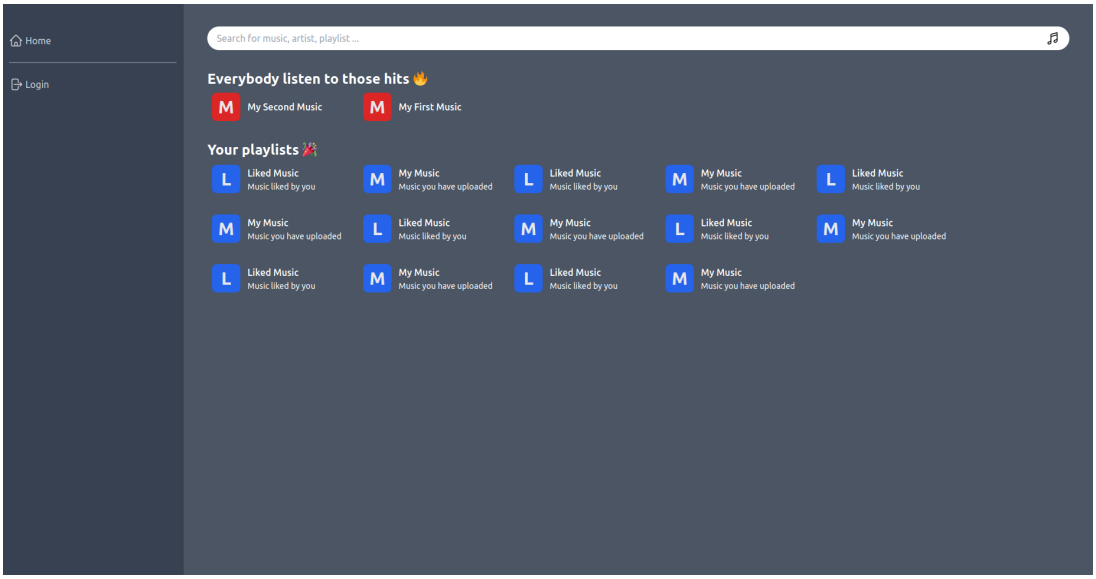

Figure 2: Screenshot of Our Music; own work

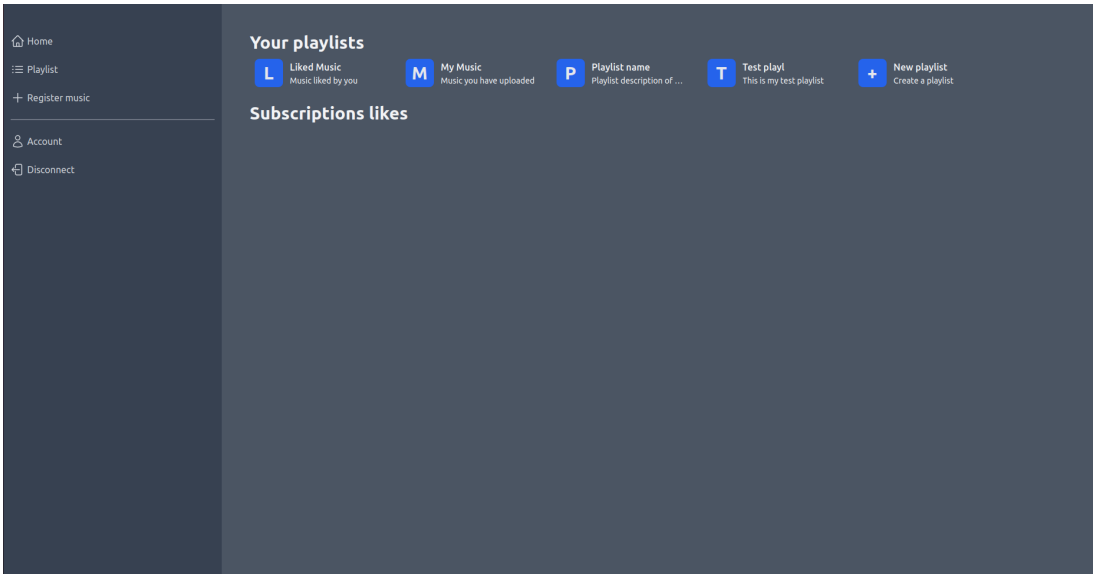

Figure 3: Screenshot of Playlists; own work

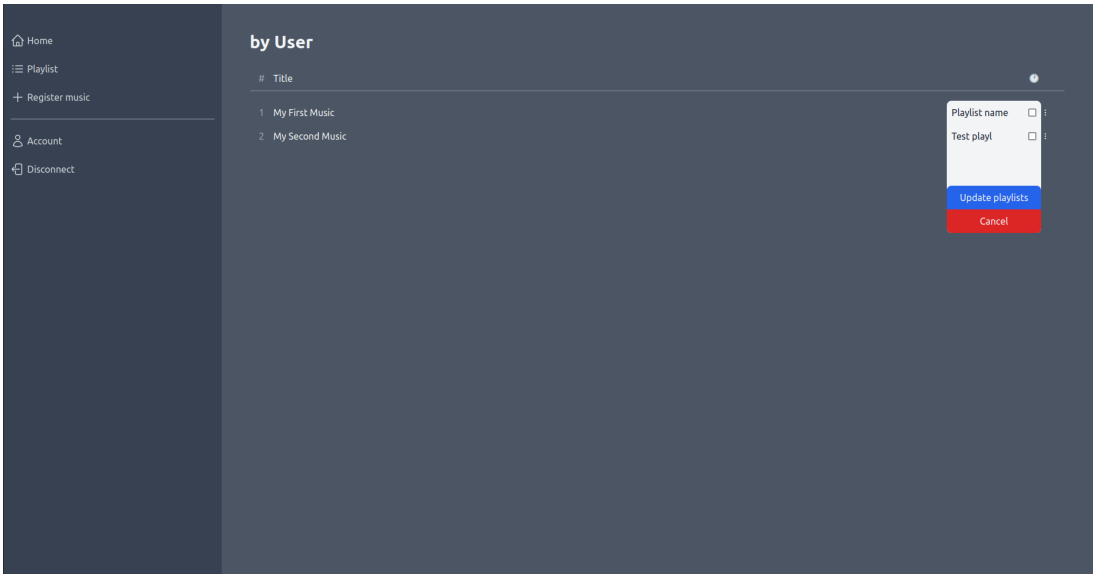

Figure 4: Screenshot of Updating Playlists; own work

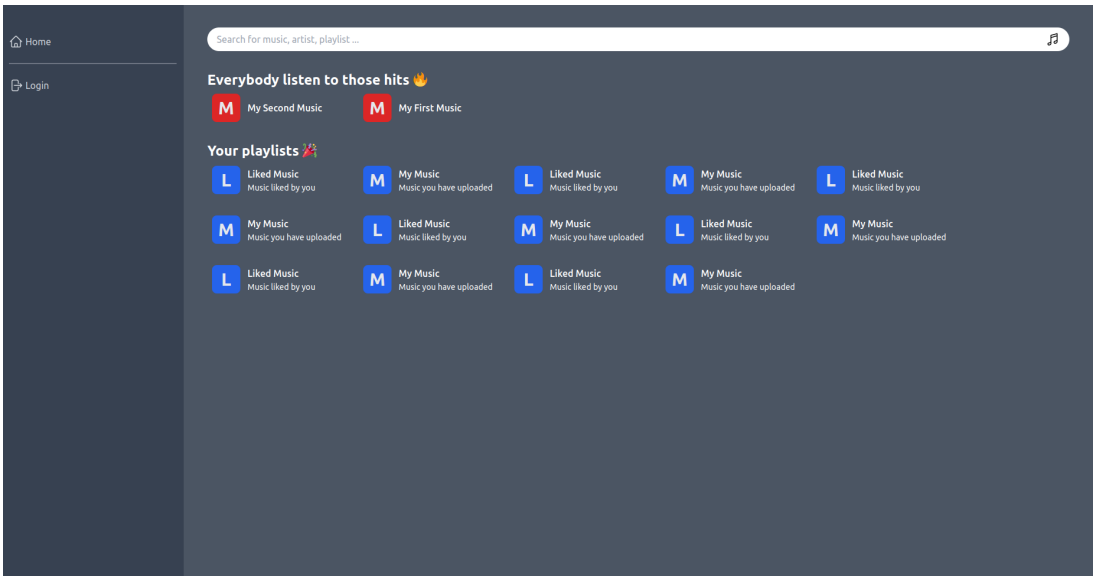

Figure 5: Screenshot of Our Music; own work

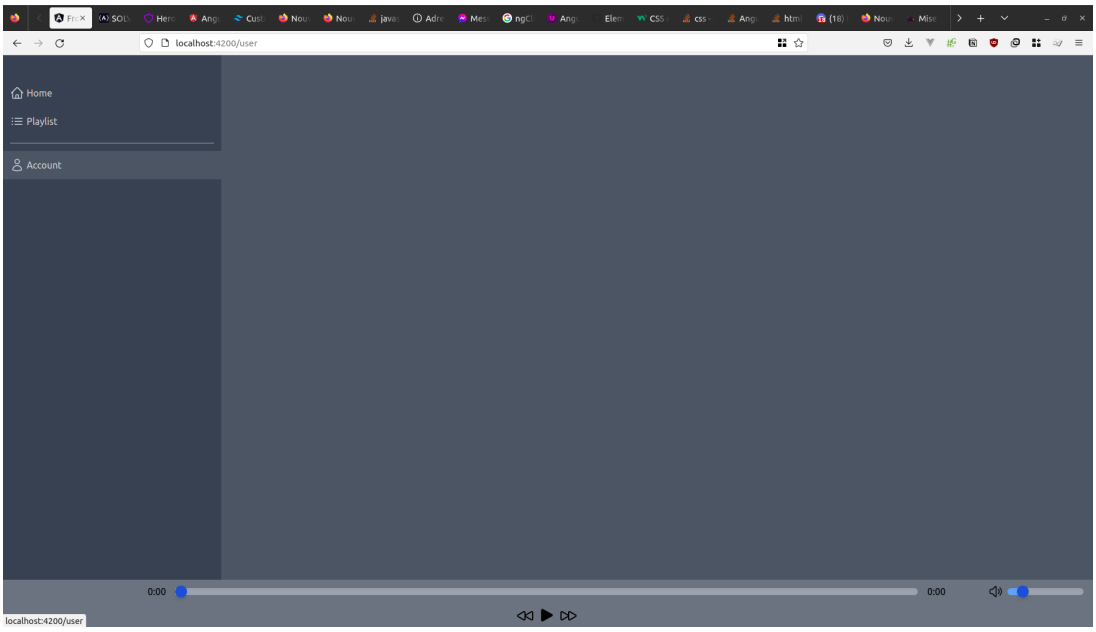

Figure 6: Screenshot of Playing Music; own work

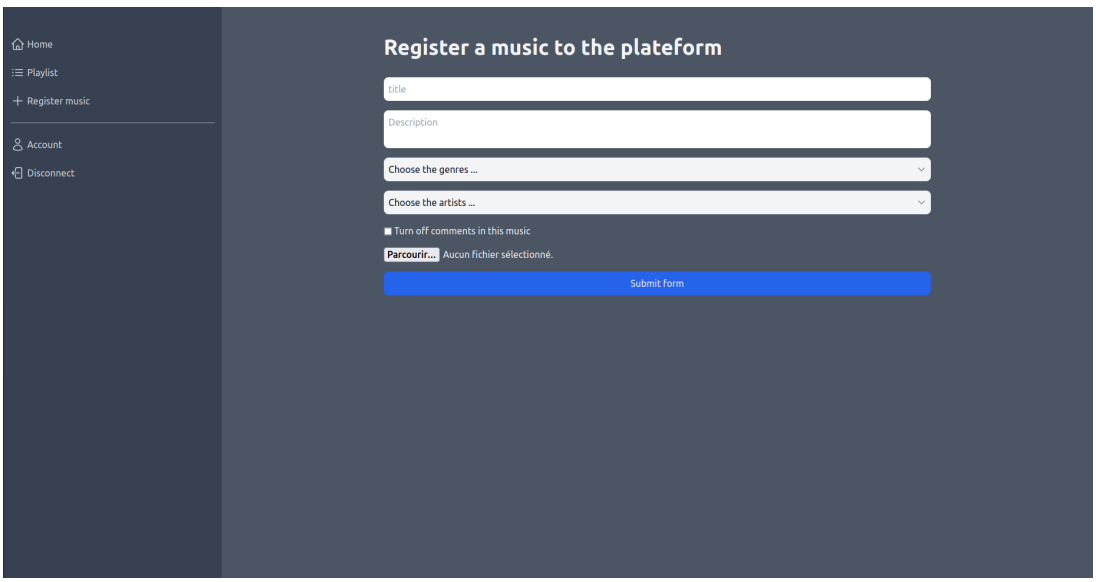

Figure 7: Screenshot of Uploading Music File; own work

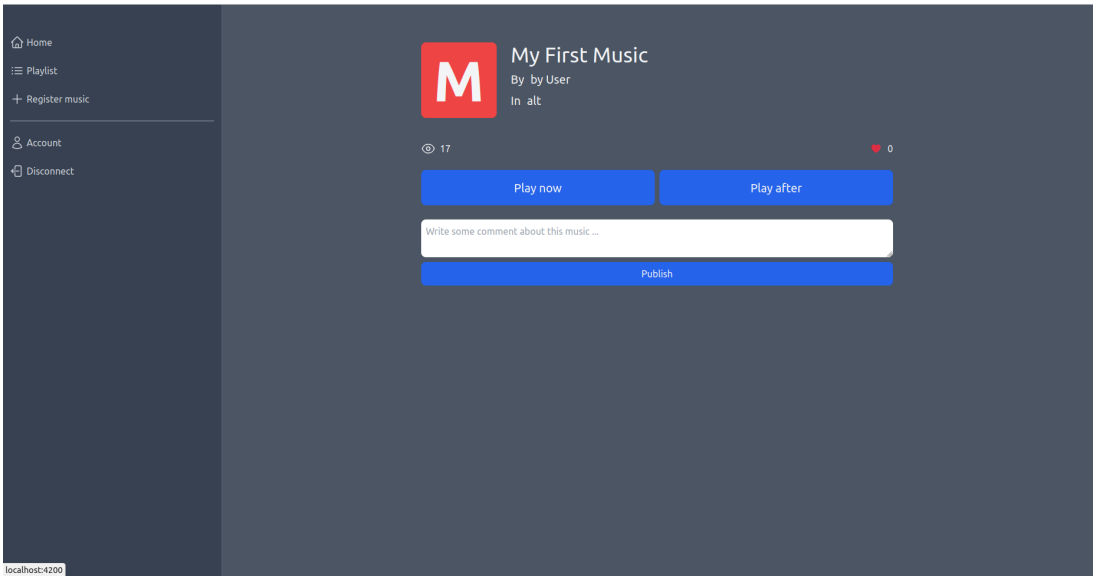

Figure 8: Screenshot of Commenting on Music; own work

Figure 9: Screenshot of User and Subscriptions; own work

6. Conclusion

The project of the music streaming app was implemented by using PostgreSQL, Kafka, ksql, NestJS and Angular, and for mocking data Python scripts were made and used. We've made a detailed ER database model and described the tables used in the model. PostgreSQL was used to implement the static part of the application along with three triggers, while we used Ksql and Kafka to stream data like views, likes and comments. The application plays music and is populated with data to demonstrate its working. For the front end Angular was used, while the back end is made with NestJS.

The application works fine, though it wasn't implemented in its entirety. For instance, we wanted to stream more data. The limitations of the technologies used in this project were tied to the connection of ksql, PostgreSQL and Kafka with the back end and front end since the connectors are hard to get by and implement, and there is limited documentation for that subject.

The application is targeted towards anyone who wants to listen to music. The app streams music by dividing the mp3 files into chunks and then buffering to play them and offers a variety of technologies for the user to use.

Bibliography

[1] "Postgresql," The PostgreSQL Global Development Group. (2023), [Online]. Available: https://www.postgresql.org/ (visited on 01/08/2023).

[2] "Apache kafka," Apache Software Foundation. (2023), [Online]. Available: https://kafka.apache.org/ (visited on 01/08/2023).

[3] "Ksqldb," Apache Software Foundation. (2023), [Online]. Available: https://ksqldb.io/ (visited on 01/08/2023).

[4] "Python," Python Software Foundation. (2023), [Online]. Available: https://www.python.org/ (visited on 01/08/2023).

[5] "Angular," (2023), [Online]. Available: https://angular.io/ (visited on 01/08/2023).

[6] N. Jatana, S. Puri, M. Ahuja, I. Kathuria, and D. Gosain, "A survey and comparison of relational and non-relational database," *International Journal of Engineering Research & Technology*, vol. 1, no. 6, pp. 1–5, 2012.

[7] H. Vyawahare, P. P. Karde, and V. M. Thakare, "A hybrid database approach using graph and relational database," in *2018 International Conference on Research in Intelligent and Computing in Engineering (RICE)*, IEEE, 2018, pp. 1–4.

List of Figures

List of Tables

List of Listings