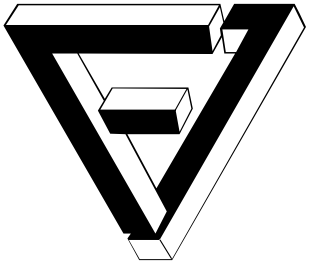
Masarykova univerzita
fakulta informatiky



Musikk. A music streaming platform with social features.

Bachelor's Thesis

Kirill Vorozhtsov

Brno, 2025

Declaration

Hereby I declare that this thesis is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.
During the preparation of this thesis, the following AI tools were used:

- **ChatGPT**: Used for debugging and making small code corrections, as well as LaTeX formatting.
- **V0**: Used for the initial styling configuration of Tailwind CSS.

I declare that they were used in accordance with the principles of academic integrity.
I checked the content and took full responsibility for it.

Thesis Advisor: Mgr. Luděk Bártek, Ph.D

Abstract

This bachelor's thesis implements a WEB-based music streaming platform with additional social features - live comment sections for playlists, user feed, additional possibilities for interaction with the followed users etc.

A study is made beforehand in order to determine the relevancy of the topic; comparison and exploration of different existing platforms is presented in order to give a better insight into the market of similar solutions.

The thesis leverages existing backend and frontend frameworks, such as Django and React, for the actual handling of the underlying data, logical processes and the interface of the platform. In addition, modern audio representation and streaming technologies, such as MPEG-DASH and HLS are used. In order for the application to feel responsive, Server Sent Events are added to provide two-way communication between the client and the server, ensuring that individual interactions are always synchronized between users and different instances of the program.

Keywords

Audio Streaming, Python, Django, React, MPEG-DASH, SSE

Contents

Chapter 1

Introduction

In recent years, with rapid development of the Internet, music has become an even more integral part of everyday life.[**music_role_life**] It has never been easier to experience and share music — we have come a long way from sharing vinyl records to simply sending a link to a streaming platform of choice. Consequently, music has integrated even deeper into social interactions between people, helping them bond and share strong emotional experiences.[**music_role_life**]
One of the direct impacts of this trend is the fast emergence of numerous music-related platforms. While some focus on traditional music journalism or statistics, others offer unlimited access to audio content. Naturally, people have started to discover and engage with music that resonates with them more frequently.[**music_role_life**]
Despite this, it is surprising that features which facilitate social interactions are not widely implemented in the existing platforms, as will be shown in Existing Platforms
The goal of this thesis is to design a music-centric platform that embraces collaboration and social interaction around music.
This work is divided into the following **six** chapters:

1. **Music Consumption Survey** Presents the outcomes of a survey illustrating how people consume music, how prevalent it is in social interactions, and why this thesis is relevant.

2. **Existing Platforms** Compares existing streaming solutions, music-related services and explores relevant non-musical platforms.

3. **??** Outlines the functional and non-functional criteria for the application.

4. **Implementation Planning** Describes the choices of technologies that are used by the application.

5. **??** Explains the development process and implementation details.

6. **Summary and Conclusion** Summarizes the results and discusses possible improvements.

Chapter 2

Music Consumption Survey

In order to better rationalize the topic of the thesis, and show that a music streaming platform centered around social interactions could be relevant as a service, a brief survey was conducted. It examined the individual content-consumption preferences, listening and discovery habits, platform usage patterns, and social behaviours.

2.1 Methods

The platform chosen for the questionnaire was 'Google Forms'[**googleforms**], as it provides a simple interface for survey creation, allows for easy sharing of the form, and supports exporting the results to a spreadsheet.

The questionnaire itself consisted of 15 questions. Most of them are multi-choice and closed-ended, with some having a possibility for a custom answer. Custom answers are grouped under the "other" answer in the provided figures. Answers in their original form can be found in the original survey.

As for the respondents, 119 people participated, most of whom were from Russian-speaking countries. The majority were in the 18 to 30 age group, with the exact distribution shown in Figure 2.1.

2.2 Results

This section presents the outcomes of the survey and analyzes their meaning. Not everything is discussed, for example, questions dedicated to music discovery are omitted.

2.2.1 Engagement

Firstly, it was necessary to determine the actual frequency of engagement with audio content – if the numbers were low, it would indicate that a

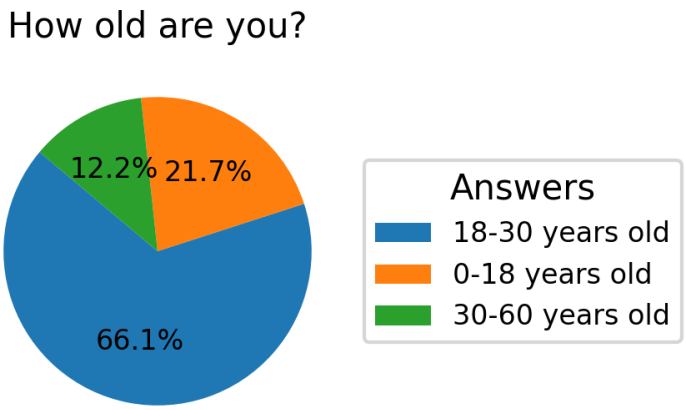


Figure 2.1: Age distribution of respondents

dedicated platform with advanced features might not be relevant for most users. However, over 50% of respondents reported listening to more than 500 songs per month (Figure 2.2), and nearly 80% stated that they listen to music daily (Figure 2.3). This clearly shows that music is essential to many people, and the following logical step was to discover how the audio content is mostly accessed.

2.2.2 Listening Methods

As can be seen in Figure 2.4, the percentage of streaming platforms usage across all ages is nearing 100%, with slightly higher numbers in lower age groups. Although downloaded files and physical media have their presence, especially for people aged 30-60, it is usually only an auxiliary option next to the streaming solutions.

The question dedicated to the popularity of individual platforms has shown that Spotify is the most used one, which is in line with the global statistics[**spotifypopularity**]. However, Yandex Music, being in second place, deserves an explanation. As mentioned previously, most of the respondents are from Russian-speaking countries, specifically Russia. With many western companies leaving its market in 2022, music streaming services included, most users have moved to locally available products - Yandex Music, VK Music, Zvuk, and others. Another notable point is the absence of other popular region-specific platforms, such as Amazon Music for the US market, QQ music for the Chinese market and JioSaavn for the Indian one, as all of the respondents were based in the European part of the world. The complete statistics are shown in Figure 2.5.

The survey showed that streaming services are indeed widely used and

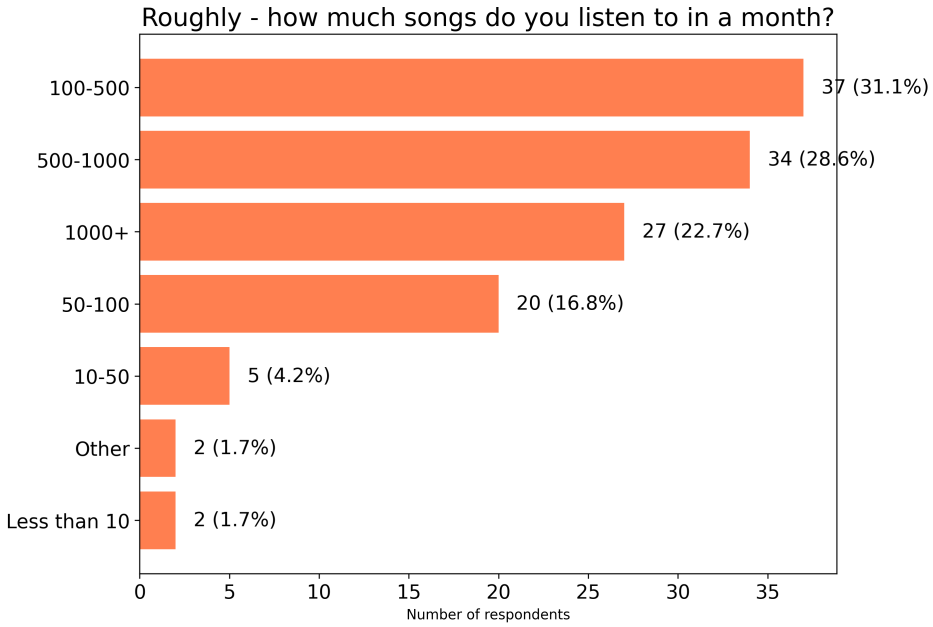


Figure 2.2: Monthly song count per respondent

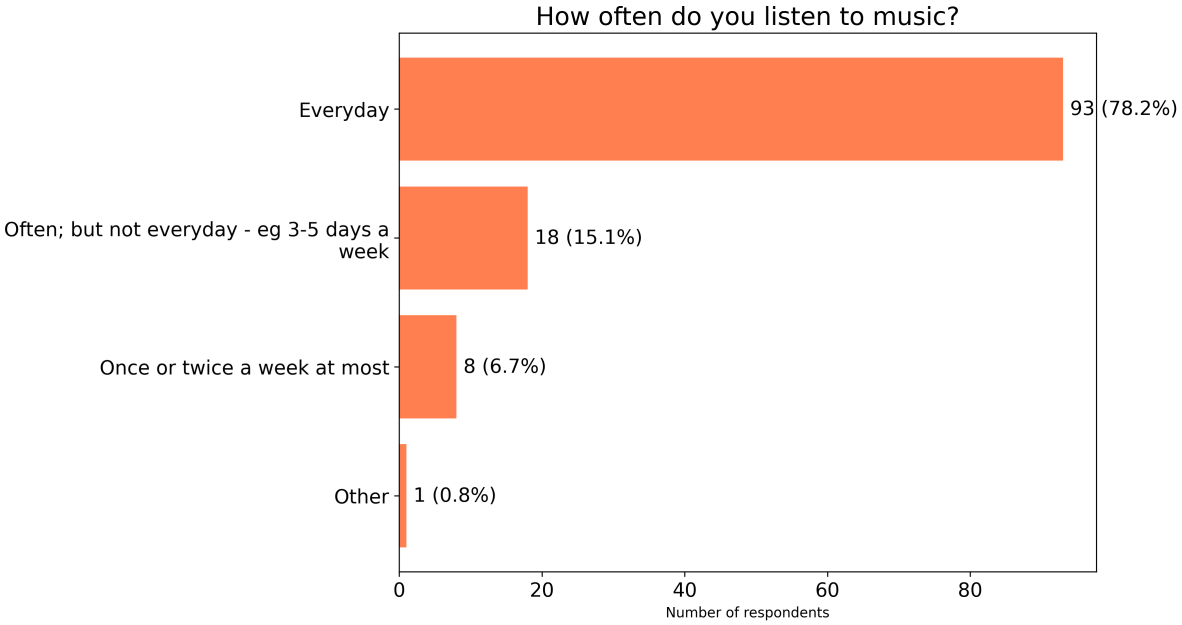


Figure 2.3: Listening frequency of respondents

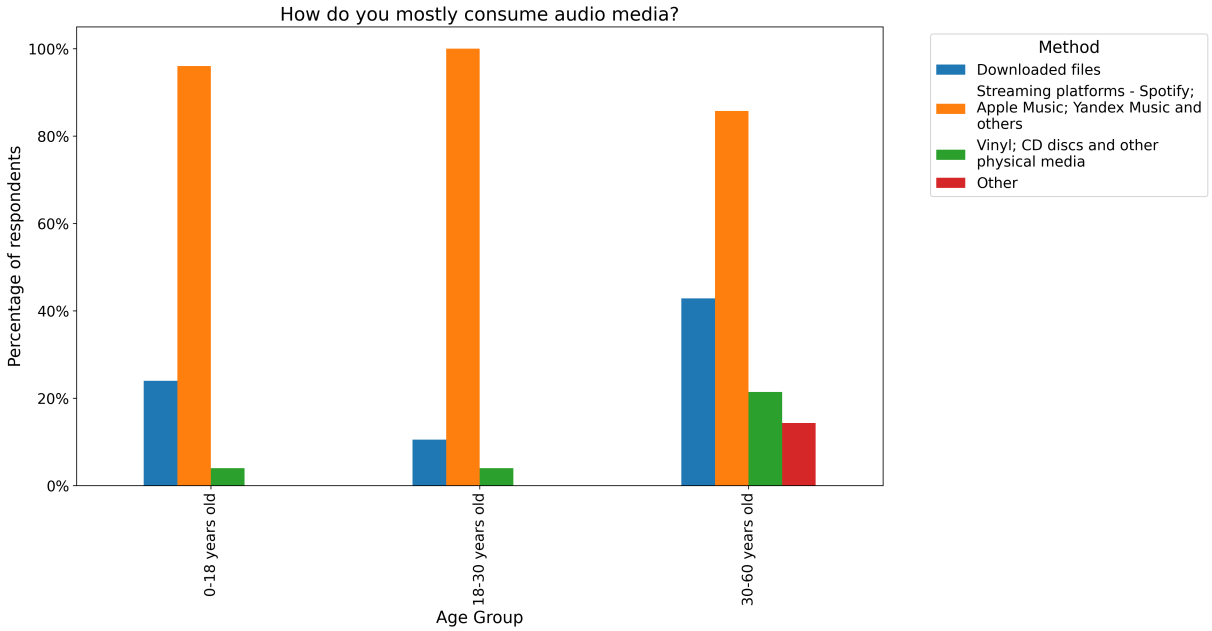


Figure 2.4: Audio media consumption by age group

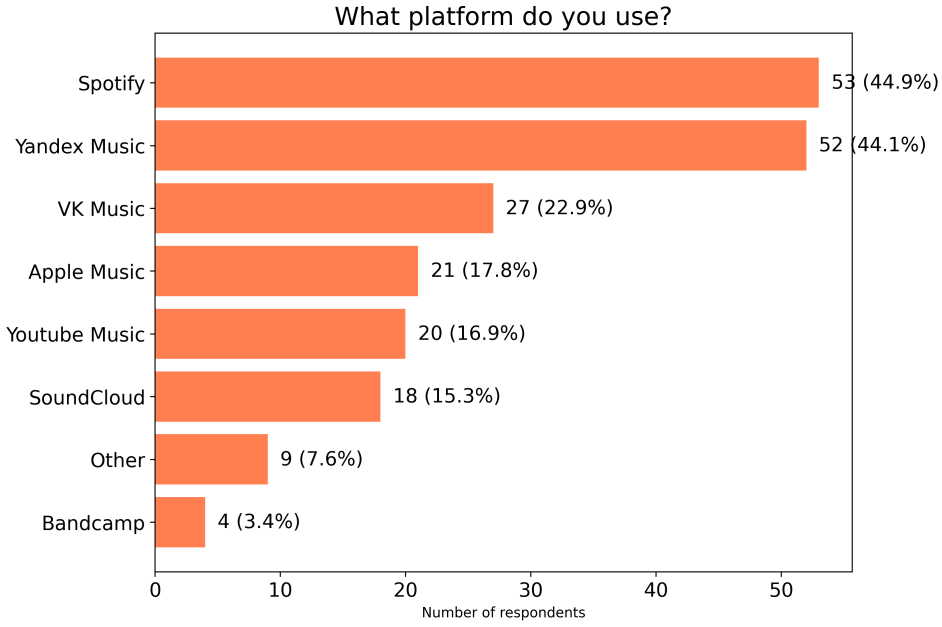


Figure 2.5: Streaming platforms popularity

in demand, the next important step was to analyze the regularity of music-centered social interactions.

2.2.3 Social Interactions

The results of Question 10 indicate that nearly all respondents (99.2%) reported listening to music with others, primarily in offline settings such as gatherings or car rides (Figure 2.6). Online co-listening methods, while less common, were also mentioned by a quarter of respondents, showing that digital solutions for shared listening are used, although not as prevalently as in-person scenarios.

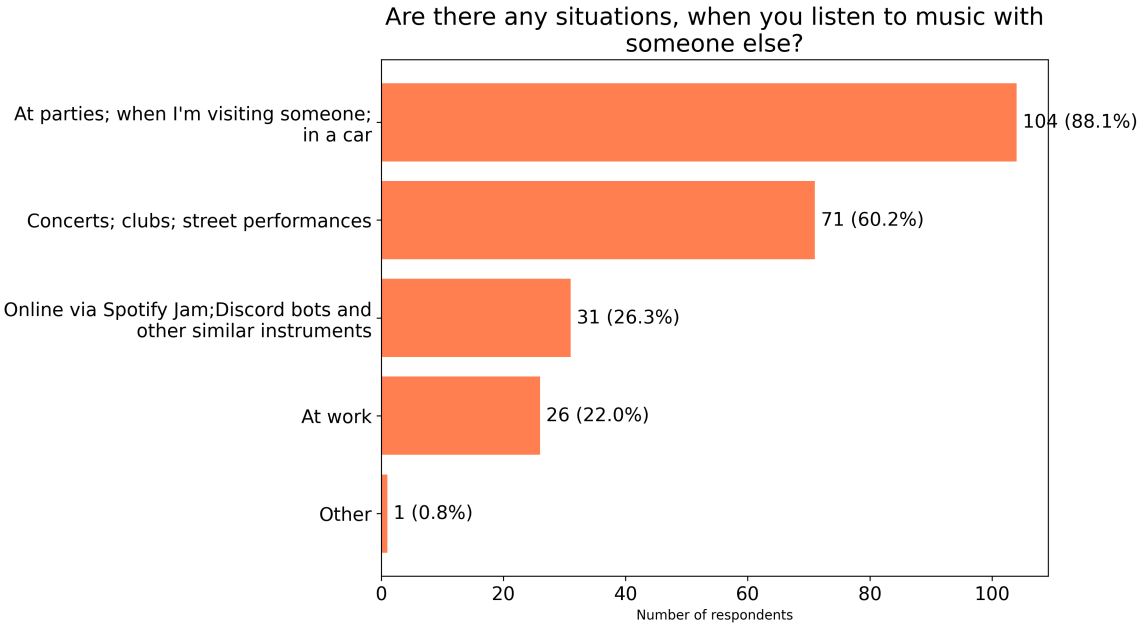


Figure 2.6: Social interaction methods

Question 11 (Figure 2.7) investigates how often these interactions occur — results show that a significant portion of respondents engage in shared listening regularly. Around 60% reported doing so at least a few times per month, with a smaller group indicating almost daily interactions. This reinforces the idea that music consumption is a common social activity.

The next question is phrased as follows: "How often do you/your friends share/discuss music?". While the previous question focused on real-time in-person collective listening, this one highlights more intentional and in-depth musical interactions, such as sharing songs, recommending music, or discussing it. Music, in that case, is not just the background but the main subject of engagement. The same trend as before can be observed in the responses: the majority of participants reported engaging in these exchanges

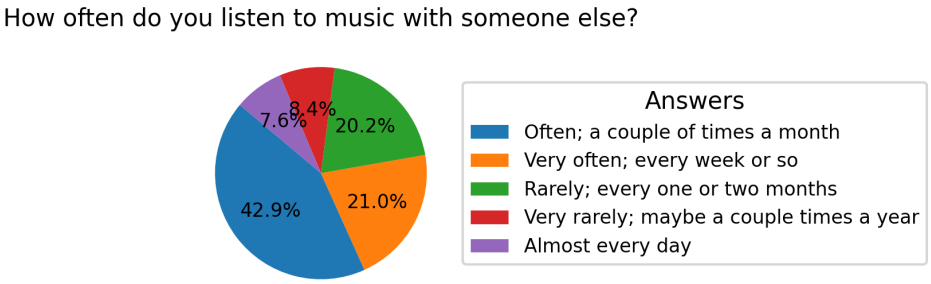


Figure 2.7: Social interactions frequency

regularly. Over 70% (Figure 2.8) indicated that they share music at least several times a month, with many doing so weekly or more often. This further emphasizes the active social role of music.

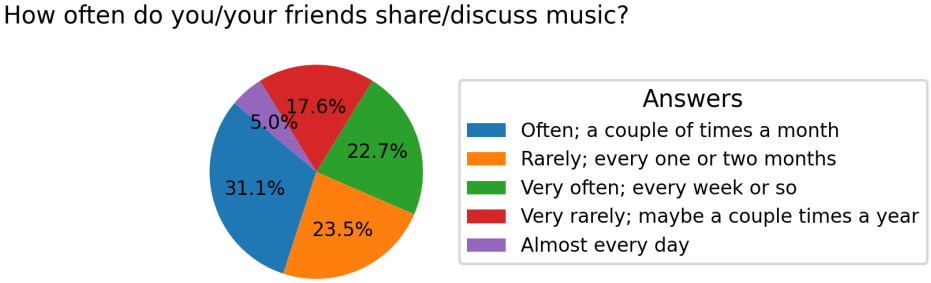


Figure 2.8: Music sharing frequency

Question 13 — "How does that happen?" (as in music sharing) — provides an insight into the specific means of sharing. Many respondents (Figure 2.9) stated that they typically send links from streaming platforms through external messaging apps. While this shows an apparent demand for sharing music online, it also reveals a gap: these interactions are fragmented across platforms. Hence, enabling such exchanges directly within the streaming app could offer a smoother, more uniform experience.

Responses to the question *"Would you say that you know a lot of people with similar music taste?"* were mixed, with a near-even split. This suggests that while some users already have a social circle with similar music taste, many do not. Interestingly, when asked *"Would you like to connect with others who share your musical taste, or influence those around you to explore and appreciate the music you enjoy?"*, the majority answered positively. This indicates an interest in the expansion of musical connections and supports the idea that social discovery features could be useful within a

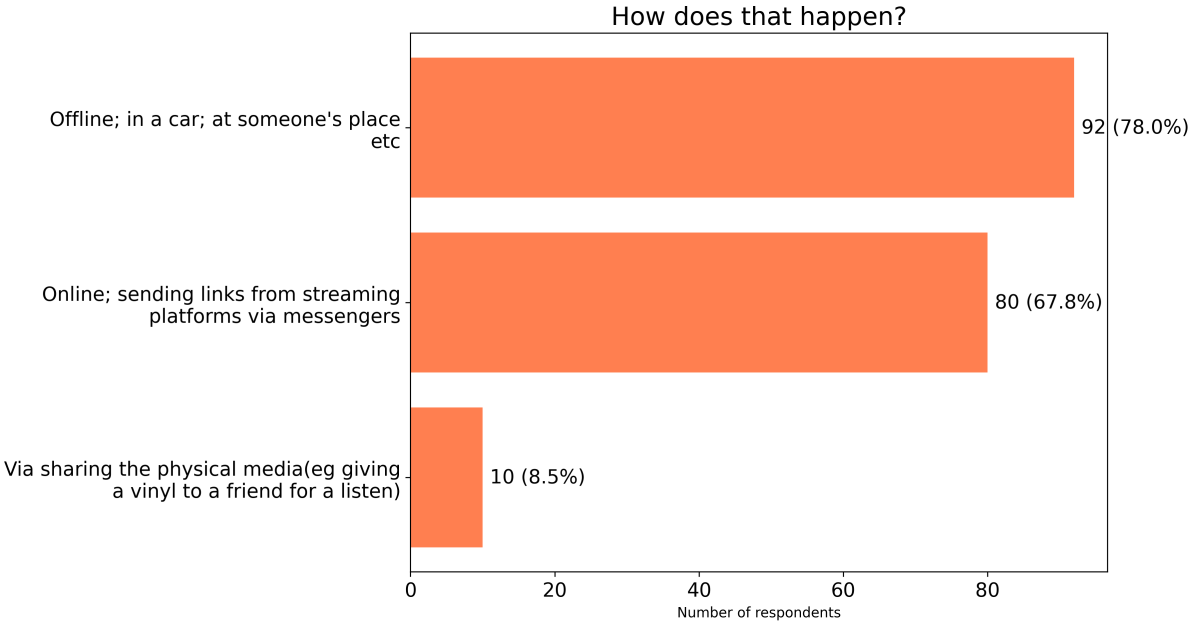


Figure 2.9: Music sharing methods

streaming platform. This is further supported by the responses to Question 15, where over 60% of participants stated they would use additional social features, if available on a streaming platform.

2.3 Summary

Overall, the results show that music is a vital part of many people's lives and that streaming services are the main way people listen to it. It is easy to see that music consumption is frequent and often appears in social interactions. Yet, current platforms only partly support interactions with other users. Therefore, the idea of a music streaming service with integrated social features could be relevant, addressing real user needs.

The full survey form can be accessed at the following link:
https://docs.google.com/forms/d/1vhhAu_SfuHV4xy6JoDaRsM5uCElYn_csTmN8u4ZlpHc

Chapter 3

Existing Platforms

To better understand what instruments people use when interacting with music, it is helpful to examine existing solutions. Additionally, this research will provide insight into the possible features to be implemented in the resulting application.

Firstly, streaming services are discussed, as the main aim of the thesis is to create a system for music streaming. Then, other music-related portals are listed, as they offer alternative possibilities to engage with audio content, which can also be beneficial for the future platform. And lastly, products such as YouTube and TikTok are discussed, as they have proven to be a popular way to interact with music as shown in 3.1.

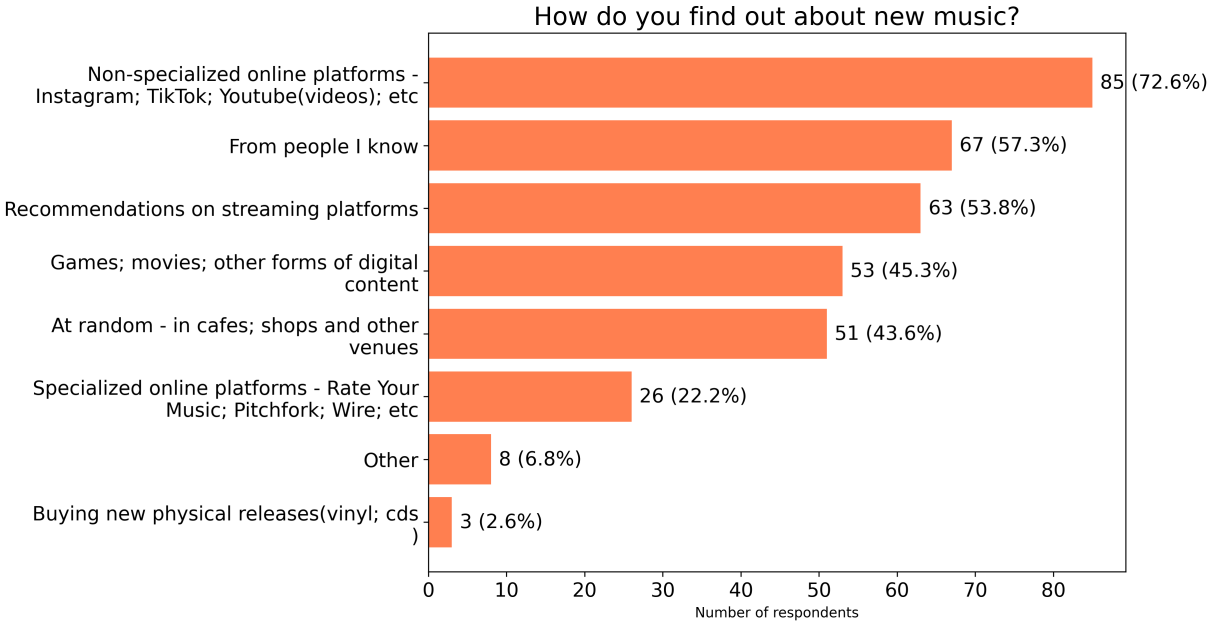


Figure 3.1: Streaming platforms popularity

3.1 Streaming Services

As can be seen in the 2.4 and further confirmed by the recent study of the International Federation of the Phonographic Industry[**music_stats_2024**], nowadays, the most prevalent way of music consumption and discovery is through streaming services. Many options are available, but only those that are both popular and have unique elements will be considered in this thesis. The descriptions, rather than offering a general portrayal of each platform, will focus on the service-specific features that include social elements.

3.1.1 Spotify

One of the most prominent social features on Spotify is the 'Spotify Jam'[**spotify_jam**]. It lets people create a collective song queue that is then synchronized among all connected users. Moreover, volume, the order of songs, and other aspects of the playback can be controlled individually. Another notable tool is the 'Blend' playlists[**spotify_recs**]. These are playlists created automatically between two people, which contain songs matching the audio preferences of both users. Lastly, 'Friend Activity'[**spotify_friend_activ**], which shows what the people you follow are currently listening to, and 'Listening Parties', which are live chats with a limited capacity, that can be joined for a short time when new music is being released[**spotify_party_1**, **spotify_party_2**].

3.1.2 VK Music

As mentioned previously, this is a music service integrated into the VK social network. Consequently, it is possible to send songs and playlists via private messages and add audio materials to posts in groups. VK Music also provides an 'Updates' tab, which is populated by the audio materials that any followed user or group has added to their 'Liked' feed.

3.1.3 SoundCloud

SoundCloud is one of the few platforms that lets its users leave comments and reactions on songs and playlists[**sc_comments**, **sc_reactions**]. In addition, each user has a feed consisting of personal uploads and reposts of other's content[**sc_reposts**], which is visible when visiting their profile.

3.1.4 Bandcamp

Every Bandcamp user has a profile with four tabs: 'collection', 'wishlist', 'followers', and 'following'. The most interesting feature is under the 'following' tab – users can subscribe to available genres and then specify the ones they want to showcase to others viewing their profile. The 'wishlist'

tab is also noteworthy, as Bandcamp's model blends streaming with the traditional purchases of individual releases, and in this tab the user can show what he is looking forward to listening in the future.

The table 3.1 summarizes the comparison between the features mentioned above.
Difficulty suggests the possible complexity of implementation.
Mode assumes the typical context in which the feature operates.
Social Interaction Level indicates the level of user involvement when using the feature.

Table 3.1: Comparison of Service-Specific Social Features

Feature	**Service**	**Difficulty**	**Mode**	**Social Interaction Lev**
Spotify Jam	Spotify	High	Group	High
Blend playlists	Spotify	Medium	Pair	Low
Friend Activity	Spotify	Low	Pair	Medium
Listening Parties	Spotify	Medium	Group	Medium
Music sending via chats	VK Music	Medium	Individual/Group	High
Embedding into posts	VK Music	Medium	Individual/Group	Medium
Updates tab	VK Music	Low	Individual	High
Comments	SoundCloud	Low	Group	High
Reposts feed	SoundCloud	Low	Individual	Medium
Genre following	Bandcamp	Medium	Individual	Low
Wishlist	Bandcamp	Low	Individual	Low

3.2 Forums, Blogs and other Music Media

A lot of different resources are gathered under these umbrella terms – from professional music review websites to amateur, personal pages. Again, only widely-used services that offer something remarkable are listed.

3.2.1 Rate Your Music

> 'Rate Your Music is one of the largest music databases online. It is an incredible tool that can help you find and learn about new music to listen to.'[**ryt**].

This website is one of the biggest platforms with user-created content. Every member is able to write reviews and set ratings for music releases, contribute to the extensive 'wiki' consisting of genres, thematic lists and charts, and

connect to other members of the forum. However, the content is still moderated – any post has to be properly formatted and abide to the guidelines in order to appear on the website.

3.2.2 2step.ru

2step.ru is one of the better representatives of 'old school' forums that is still active. This is a country-specific resource, and as a consequence, on top of providing the regular music and forum components, there are elements usually missing from bigger portals. For example, a section about upcoming parties, a page dedicated to local and upcoming DJs, and an ongoing list of recorded radio shows.[**2step**]

3.2.3 last.fm

Last.fm is a music discovery and social networking service built around its 'scrobbling' feature, which automatically records every track played through connected players to the user's profile, creating an exhaustive listening history[**lastfm**]. Based on this data, Last.fm generates personalized recommendations and charts, while tag-based navigation enables exploration of genres and user-curated collections[**lastfm_tags**]. Social interaction is fostered through friend lists, public groups for discussion, and the ability to comment on artist and track pages. Additionally, an 'Events' section aggregates concert listings and allows users to indicate that they are interested or confirm that they are going and to add comments under each event post, bridging online listening activity with real-world live music experiences[**lastfm_events**].

In summary, these platforms illustrate the range of community-driven and data-driven approaches to music engagement beyond pure streaming. They highlight how social interaction and user-generated content can enrich interactions with music-related content.

3.3 Other Platforms

As mentioned earlier, one of the most interesting results of the survey is the high number of people discovering music on platforms that are not focused on music. Services like Instagram, YouTube, and TikTok—mainly used for sharing videos and other user content—have become closely connected to how people find and listen to music. In recent years, these platforms have started to promote music more directly by making it part of short, easily shareable content that spreads quickly. Instagram, for example, added a special audio tool that lets users include music in their 'Reels'[**inst_audio**], helping songs reach more people through visual content. YouTube automatically finds songs used in videos and shows them in the description, making finding and listening to them easier. TikTok has had an even bigger impact

– many songs become viral and later become hits in charts and streaming apps, because people use them in popular videos. These are often called "TikTok hits."

This shows that big platforms are not just reacting to users' interest in music – they actively help promote it by connecting songs with content that people want to watch and share.

And, for example, an integration with such services can be a big plus for the platform, improving the user experience when it comes to the exploration of the songs they liked.

Summary

As can be seen, there are numerous instruments available across the Internet. However, services that focus on the actual audio delivery often lack features stretching beyond that, forcing users to use multiple platforms in order to meet their needs. This project aims to bridge that gap, particularly by enhancing users' potential for social interaction.

Based on the research done in the previous and current chapters, these social features were chosen for implementation, as they are both easy to implement and provide a high level of social interaction:

- Comments under audio content.
- Individual posts and a feed with posts of others.
- Friend listening activity.
- Latest content added by friends.

These and further requirements for the application are listed in the following chapter - Specification.

Chapter 4

Specification

In this chapter the general outline of the application is specified via functional and non-functional requirements. The individual requirements are constructed based on the 'MoSCoW' criteria[**moscow**].

4.1 Important Terms

Below is the list of important terms that will be appearing throughout the requirements section.

- **Anonymous User** – A User which is not yet registered in the system or has not logged in.
- **User Profile** – An object which encompasses all the information related to a specific User.
- **Song** – An object representing an individual audio object.
- **Song Collection** – Represents a container that holds multiple Songs.
- **Playback** – A process during which the user is receiving audio information.
- **Playback Item** – A Song or a Song Collection.
- **Playback Queue** – A list of Playback Items.
- **Comment** – A small piece of text provided by the User input usually placed under a specific object to which it refers.
- **Reply Comment** – The same as Comment, but must be created in relation to another, 'parent' Comment.

4.2 Functional Requirements

Requirement	Priority
The system shall support over-the-net audio Playback.	Must Have
The system shall provide a way to control the audio Playback. That includes shifting the Playback backward and forward, stopping the playback.	Must Have
The system shall provide a way to show Playback Items.	Must Have
The system shall support new Playback Item addition.	Must Have
The system shall provide a way to enqueue Playback Items and the means to manipulate the Playback Queue.	Must Have
The system shall have individual User Profiles.	Must Have
The system shall allow Anonymous Users to create a User Profile and log in to the system using email and password.	Must Have
The system shall allow Users to change their User Profile information.	Must Have
The system shall differentiate presented content based on the User. That includes Songs and Song Collections, User Profile information and other related items.	Must Have
The system shall allow for Song and Song Collection creation. Only Artists shall be allowed to create Songs.	Must Have
The system shall provide a way for Users to add Comments under Song Collections	Must Have
The system shall provide a Forum system. Forums shall be able to be created by Users. Forum Comments must be able to embed Playback Items.	Must Have
The system shall provide a way for Users to create relations to other Users. All of the Comment-related systems shall be able to be filtered on these User relations.	Must Have
The system shall have a way to filter and search Playback Items, Users and Content Owners.	Must Have
The system shall make it possible to create Reply Comments for Comments.	Should Have
The system shall provide a notification system. It shall include Reply Comments and other appropriate items.	Should Have
The system shall make it possible to filter Playback Items based on the User relations.	Should Have
The system shall provide chat capabilities.	Could Have

Table 4.1: Functional Requirements

4.3 Non-functional Requirements

Requirement	**Priority**
The system shall provide access to its contents only to authorized Users. The only exception shall be the 'Log In/Sign Up' page.	Must Have
The system shall store content securely; data representation shall be as efficient as possible.	Must Have
The system shall store and transport sensible user information only in safe manners.	Must Have
The system shall, in case of failure, remain rendered on the screen and show the appropriate non-technical message to the User.	Must Have
The system shall render new available information without refreshing the content page. That includes new Comments, Playback Items, updated Playback Queue and other related items.	Must Have
The system shall provide a responsive search method which would react to dynamic user input.	Should Have
The system shall synchronize Playback across multiple devices.	Should Have

Table 4.2: Non-functional Requirements

Chapter 5

Implementation Planning

This section explains the high-level technology choices behind the application. The following areas are discussed, as they have the most significant influence on the overall implementation of the service: Backend, Frontend, audio transfer, authentication, and server-client communication.

5.1 Backend

Python[**python**] was chosen as the primary language for this part of the application. It supports all the needed instruments, such as external process calls, file handling, and asynchronous code execution. Moreover, it has a flexible and straightforward syntax allowing for rapid development. As this project is mostly IO-driven and most of the processor-heavy operations are offloaded to external processes, i.e., no thread programming is used, Python's performance limitations[**gil**] will not have such a drastic effect.

In order to avoid errors related to building the system from the ground up and shorten the development time, it was decided that a framework would be used.

5.1.1 Framework Comparison

Currently, there are three well-developed and widely used frameworks available for Python:

- **FastAPI**
  FastAPI is a comparably quick framework that can be on par in terms of speed with frameworks from other languages, such as NodeJS or Go [**fastapi**]. As the official website states:

  > 'FastAPI is a modern, fast (high-performance), web framework for building APIs with Python, based on standard

Python type hints.' [**fastapi**].

On top of that, it offers full integration with API documentation standards such as Open API, and API representation tools, e.g., Swagger.

However, it is relatively new and basic; given the requirements and the fact that few additional packages are available for this framework - lots of custom code would have to be written on top, which is a disadvantage for this specific project.

- **Django**
  Django is a 'batteries-included' framework, in the sense that almost everything - from access control and database management to API routing and the admin interface - is a part of it.[**django**] Additionally, Django offers an extensive ecosystem, including support for REST APIs through `django-rest-framework`[**drf**], various authentication methods, and multiple data transfer protocols. Even two-way communication, such as WebSockets and Server-Sent Events, is supported via the `django-channels`[**django_channels**] package.

  Some notable drawbacks include its opinionated design choices, which can make it hard to add custom logic. In addition, the size and speed of the application can be worse than in the FastAPI case, as this framework includes many parts that may not be used but still bloat and slow down the system.

- **Flask**
  Flask is a micro-framework built on the WSGI interface. It is well-suited for smaller applications and follows a minimalist philosophy when selecting the components that are used.[**flask**] Most additional functionality is provided by the community and is distributed as external packages.

  One of the main downsides of Flask is that it does not integrate well enough with the ASGI interface, which is necessary for the proper handling of the Server-Client communication part. Moreover, strongly relying on external packages can pose potential security risks.

Of the three frameworks, Django is the most suitable one. Most of the application logic will be handled by the backend, heavily utilizing Django's ORM for most database tasks, simplifying the development. A separate library would have been needed for other frameworks, such as SQLAlchemy[**sqlalchemy**]. On top of that, a JSON-based API could be implemented using 'django-rest-framework', which would significantly reduce the amount of boilerplate code needed for the application. And lastly, Django has the biggest community acceptance, which can be verified, for example, by looking at the StackOverflow[**stackoverflow**] search results for individual frameworks.[**so_fastapi**],[**so_django**],[s

Having established what will be used for the backend application logic, the database had to be chosen next.

5.1.2 Database

Below is the comparison of the databases that the Django framework supports. It is worth mentioning that two other databases can be used as well - Oracle and MySQL. However, Oracle is ruled out due to its commercial license, and MySQL is not considered because MariaDB is fully compatible with it, while providing multiple enhancements.

- **SQLite**

  > 'SQLite is an embedded SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files.'[**sqlite**].

  As a result of SQLite's design, it is suitable in situations where simple storage is needed and write operations are limited. Moreover, as SQLite stores all information in a file on a disk, future scalability could be an issue; changing the database mid-way is often hard.

- **PostgreSQL**
  PostgreSQL is a traditional object-relational database almost entirely compatible with the SQL standard and a de facto standard in most modern applications. It offers many supplementary features, such as complex data types, proper replication and logging, full text search, etc. In addition, PostgreSQL provides extensions that can fit the database to the specific application. [**postgres**] Django has the best support for it, adding a lot of wrappers for PostgreSQL's custom logic.[**django_postgres**]

- **MariaDB**
  MariaDB is a fork of MySQL, which adds several advanced features and performance optimizations. One prominent aspect of this database is its pluggable storage engine architecture, which allows one to choose a different engine per table for the needed workload.[**mariadb**] It can also be lightweight and use fewer system resources than PostgreSQL if set up right. However, it is not fully in line with the SQL standard - e.g., limited ALTER statements and GROUP BY handling[**dbfunctions**], which often lead to unexpected time losses with debugging. And as a consequence of its lesser acceptance[**dbrank**], the community support for the non-MySQL parts of the database is not as broad.

Three main points secured PostgreSQL as the database of choice:

1. More advanced full text search functionality compared to SQLite and MariaDB.

2. Integrated Django wrappers around PostgreSQL, which greatly reduce the amount of handwritten SQL.

3. Superior syntax, command-line interface, and error handling/messages.

After establishing the backend stack, the next logical step would be choosing the frontend technologies.

5.2 Frontend

Firstly, the language had to be chosen.

Although JavaScript is the primary language for client-side development, other languages such as Rust, Python, and C can also be used to build frontend applications. However, these languages are not directly supported by the browsers, meaning the code must be compiled to WebAssembly first (supported by browsers). This adds additional overhead to the development cycle, introduces new toolchains that add another level of complexity and often results in bigger bundle(hence - download) sizes. Furthermore, these languages often lack solutions for common web development problems, compared to JavaScript, which has a lot of libraries and packages ready to be installed. That is why the application's Frontend is written in JavaScript and only JS frameworks are compared.

Modern JavaScript development is primarily dominated by three technologies: React, Vue, and Angular.[**frameworkdata**] Each introduces a different approach to building frontend applications and has its own trade-offs.

- **React**
  React is a JavaScript library focused on building user interfaces through a component-based architecture.[**react**] It emphasizes a declarative style of UI design and integrates well with various state management tools and libraries.

  One of the main React's strengths is its ecosystem - a wide range of third-party packages support it, developer tools, and documentation. Additionally, React is maintained by a large team, and many examples of working and deployed applications can be found. However, the common problem mentioned is its complexity and lack of standards, which could lead to highly inconsistent code and performance problems later in development, if not used carefully.

- **Vue**
  Vue, in comparison to React, is a framework, which means that it has most of the basic, often-used features built-in(e.g., routing or forms)[**vue**] Many state that it is easier to pick up than React, due to its more intuitive templating syntax, and built-ins mentioned before, which can lead to faster initial development.

  The main drawback of Vue is its adoption.[**frameworkdata**] Fewer packages and less documentation are available online compared to React or Angular. In some cases, it can lead to the need for the implementation of features already covered by existing solutions in other frameworks.

- **Angular**
  Angular is a full-stack framework for building web applications - hence, like Vue, it provides many tools out of the box. It is built on top of TypeScript and enforces the usage of pre-chosen architectural patterns, often leading to more robust and structured code. This is a plus for bigger projects, but it can significantly slow down the speed of progress.

While all three frameworks offer modern development patterns and are capable of being used in production applications, React was chosen. It is mature, has an extensive ecosystem, and is widely adopted. To add, it does not force many restrictions and has a good debugging interface. Since this project is not going to be large, React would possibly be the better option due to the sheer pace of development that can be achieved with it.

After giving the outline of the client and server sides of the implementation, as the main aim of the application is music delivery, the appropriate protocols had to be examined as well.

5.3 Audio Transfer

Audio playback can be tackled in numerous ways:

- **Basic Download**
  The audio file is downloaded in its entirety before playback begins. Once the download is complete, the user can listen to the file offline. This method typically requires significant storage space and time to download, especially for larger audio files.

- **Progressive Download**
  The audio file begins to play after a portion of the file has been downloaded. As the playback continues, more of the file is downloaded in the background. This method allows quicker access to the content

than basic downloading, but still requires enough data to be buffered at the start or during seeking.

- **Streaming**
  Audio is transmitted over the internet in real-time, without needing to be downloaded. The user can start listening almost immediately while the audio is being sent from a remote server. This method does not require local storage and is ideal for accessing content on demand, but it depends on a stable internet connection. Additionally, modern streaming protocols support adaptive streaming, allowing the audio file to be delivered in multiple representations. Different bitrates and codecs can be selected based on the network conditions or device capabilities, providing an optimized experience. Furthermore, seeking can be more accurate because streaming protocols divide the data into smaller, discrete chunks, allowing quicker access to specific points in the audio.

- **Peer to Peer**
  In P2P audio transfer, audio data is shared directly between users' devices, rather than being served from a central server. Each user acts as both a client and a server, contributing to and consuming data. This method can reduce server load and improve access speed, but relies heavily on user participation and network conditions.

Although basic and progressive download methods are much simpler in implementation, streaming has many valuable benefits, allowing for smooth playback in different conditions and better audio representation techniques. P2P is also not suitable in the current case, as the files must be on users' machines, which can lead to various problems: from delays for clients located far away from the source of the audio to problematic enforcement of possible DRM (anti-piracy) protection.

Another important choice would be the authentication method, as it can significantly influence both the design of the frontend and backend parts.

5.4 Authentication

There are many different methods of authentication available, but only the ones suitable for web applications are considered below:

- **Basic Authentication**
  Basic Authentication works by sending a username and password with each request, typically encoded in base64.[**basic_auth**] It is simple to configure, and it can work well for e.g., communication between internal services. However, it incorporates credentials in every request, making it prone to security risks in client-server systems if not used

over HTTPS. Moreover, it lacks session management, limiting scalability.

- **Session Authentication**
  Session authentication creates a session after the user logs in, with session data stored on the server, and the client storing a session ID in a cookie. It makes authentication management centralized, which can enhance security compared to Basic Authentication. However, it can be vulnerable to session hijacking if cookies are not properly protected, and can greatly increase the load on the database.[**session_auth**]

- **JWT (JSON Web Token)**
  JWT is a compact, URL-safe method of representing claims between two parties.[**jwt**] It is an example of stateless authentication where the token has all the needed information for user authentication. It can also carry an additional custom payload, which, for instance, can be utilized for role management. They provide a flexible and scalable way to implement secure authentication. However, if not properly managed (e.g., using long expiration times), they can become security risks, as compromised tokens may be used for extended periods.

- **OAuth 2.0**
  OAuth 2.0 is an authorization framework that allows third-party applications to access resources on behalf of the user, without exposing user credentials.[**oauth2**] It requires the user to have an account with a company that owns a resource server(e.g., Google, Facebook...). This is a proper means of authorization, however, it must be used in conjunction with other regular methods.

JWT tokens were chosen as the primary method of authentication. They are easy to set up, behave well when the application scales horizontally, and, most importantly, do not use the database to manage authentication details. Additionally, they can greatly simplify the login/logout phase and lower the number of API calls needed to get the User information, as some of the information can be embedded inside.

The last important concept that needs to be worked out is the server-client communication.

5.5 Server-Client Communication

As it was pointed out before, the communication in the application is going to be bi-directional. Because this is a web application, techniques such as Webhooks, gRPC, Pub/Sub, and others that are not directly supported by browsers will not be discussed. Traditional polling is also not mentioned due to its performance limitations and inefficiency. P2P techniques(e.g.,

WebRTC) will not be listed, because the application's architecture is client-server, not peer-to-peer.

- **Long-Polling**
  Long-polling is a technique where the client sends a request and waits until the server responds, potentially holding the request open for an extended period. Once a response is received, the client immediately issues another request. While this method works in environments where other techniques are not available, it introduces unnecessary overhead by repeatedly opening and closing HTTP connections. It also increases server load under high traffic, proving unsuitable for the logic outlined in **??**.

- **Server-Sent Events (SSE)**
  SSE is a one-way communication protocol that allows the server to push updates to the client over a single, long-lived HTTP connection. Firstly, the client establishes the connection via an HTTP request, then the server accepts and stores it, later sending special HTTP responses(technically one never-ending response) using that channel.
  It is simpler to implement, works over standard HTTP/1.1, which helps with debugging, and is supported by most modern browsers.[**sse**]
  SSE is limited to server-to-client communication, but this limitation is not critical for applications where the client initiates most of the interactions. In addition, it is easy to set up using the `django-evenstream`[**django_sse**] package that allows to integrate SSE with regular `django-rest-framework` components.

- **WebSockets**
  WebSockets provide full-duplex communication over a single persistent TCP connection. This method allows messages to be sent both ways at any moment and is optimal for real-time applications[**websockets**]. However, implementing WebSockets requires additional infrastructure and logic for managing connections and message routing. In the case of Django, support for WebSockets is realized via `django-channels` package, but all of the communication handling still has to be implemented by the developer, which leads to a prolonged development time.

  Furthermore, WebSocket traffic is not always reliably supported in all environments. Firewalls and corporate proxies may block or degrade persistent TCP connections, affecting delivery and/or availability.

- **Web Workers Push API**
  This technique allows background scripts in the browser(service workers) to receive push notifications from the server, even when the web page is not active.[**pushapi**] While useful for notification systems,

where message delivery time does not play a big role, it is not suitable for interaction that requires an immediate response.

Although WebSockets offer the most complete solution for real-time two-way communication and are well-suited for the target architecture, the complexity of their setup in Django makes them less practical in development. Therefore, Server-Sent Events were chosen as the communication method; SSE provides adequate performance and responsiveness for the expected interaction model while reducing implementation overhead.

5.6 Summary

The created platform is going to be built using the Model-View-Controller [**mvc**] architectural pattern and is split into three major parts:

- **Backend.** Implements all the data-related logic and processing.
- **API.** Is responsible for transferring the data to the Frontend and receiving commands from it.
- **Frontend.** Presents the data to the User and accepts their commands.

The next chapter Implementation provides the details of the actual implementation of the application.

Chapter 6

Implementation

This chapter presents the created platform; it is structured somewhat similarly to the previous one - firstly, the audio processing, representation and transfer are discussed, then both the Backend and Frontend are introduced, and lastly, the deployment process is shown. In addition, UML diagrams are provided as an overview of the system.

High level overview can be seen in the following 6.1, 6.2, 6.3 diagrams.

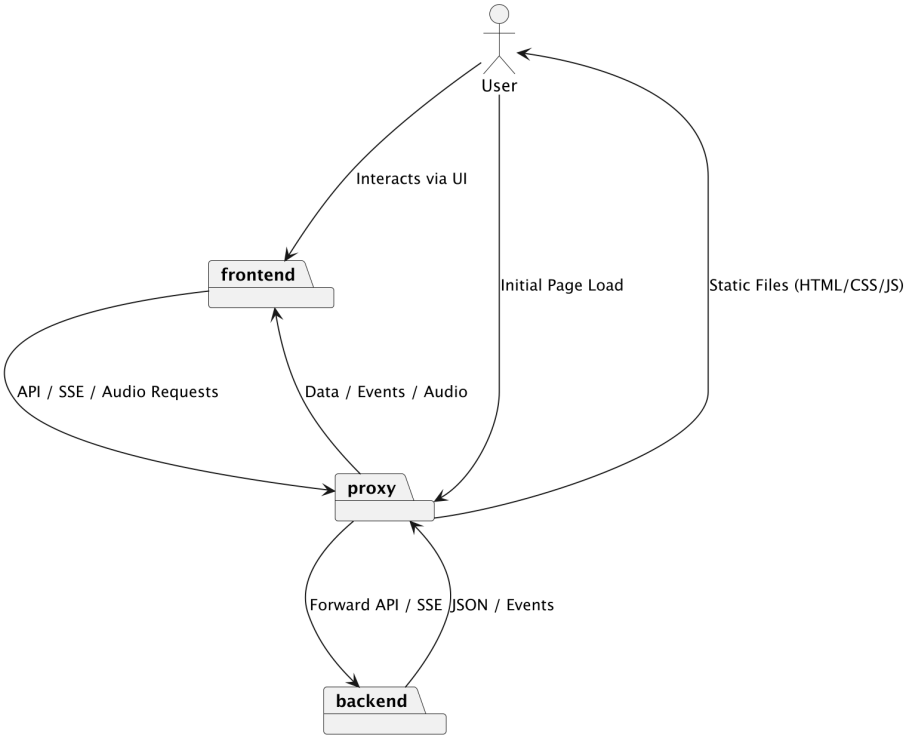


Figure 6.1: System Overview

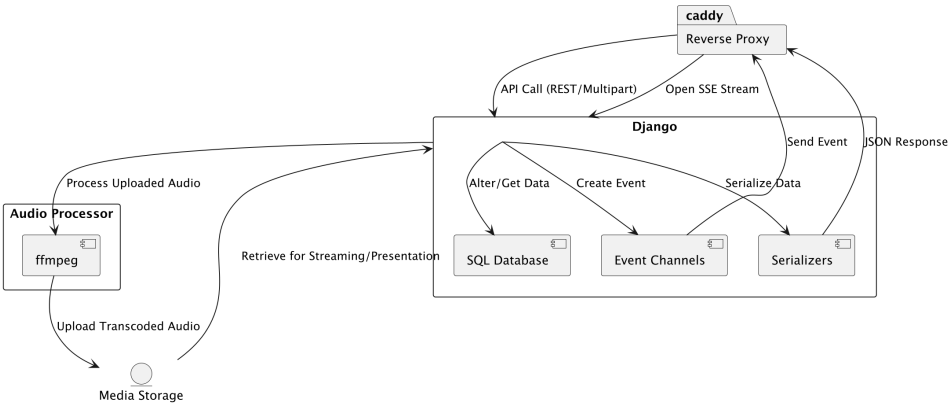


Figure 6.2: Backend

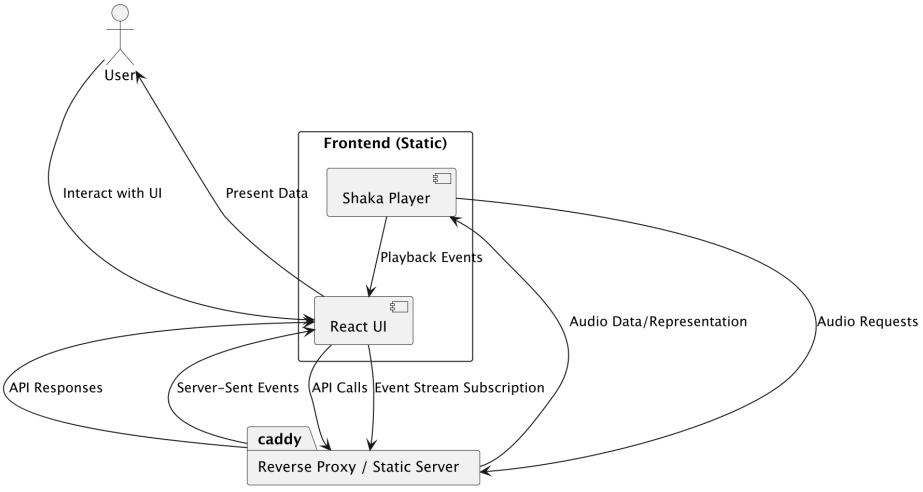


Figure 6.3: Frontend

6.1 Audio Processing, Representation and Transfer

The most important part that had to be implemented was the audio-related functionality, hence, the application development begun with it.

Below, the core aspects of handling audio are described —- beginning with the selection of streaming protocols, followed by how audio is represented and converted into streamable formats, and finally the tools and approaches used for processing the audio into the desired output suitable for web delivery.

6.1.1 Streaming Protocols

Two protocols were chosen for audio streaming - DASH[**dash**] and HLS[**hls**]. Broadly speaking, they work by breaking the initial audio file into smaller chunks and present them via a manifest, which is a listing with all available chunks separated by their timecodes. Then it is possible to use that manifest to request individual chunks instead of fetching the whole file. Moreover, it is possible to use different coders and bitrates for the initial file, which would all be arranged together in the resulting manifest, giving the possibility to choose which chunk will be served next - e.g. in the case of web audio players the next chunk can be chosen based on multiple parameters, for example, network conditions.

At first, only DASH was considered, as it is codec and container agnostic, is more efficient for lower bitrates, can stream lossless audio(HLS can only on Apple devices), has multiple DRM implementations to choose from and many further advantages. However, Safari does not support it well, and on iOS below 17.1 it does not work due to Media Source Extensions [**mse**, **msecaniuse**] not being present. Since Safari is the second most used browser [**browserusage**], HLS is used as well.

After choosing the streaming protocols, it is needed to find the way of converting audio to formats that they support. Audio representation is discussed first, with tools for the actual audio processing next.

6.1.2 Representation

Usually when audio is recorded, it is initially saved in lossless formats, which are big in size, due to the absence of any compression. For instance, the resulting WAV file for a 3-minute stereo audio recorded with bit depth of 24 bits and a sample rate of 41khz is approximately 42 megabytes, which makes it very impractical for over-the-net streaming.

Codecs are special programs which aim to reduce the size while compromising audio quality as little as possible. Most often they take in multiple input parameters which affect the resulting converted audio, among them

is bitrate; the initial bitrate for the raw recorded audio can be calculated as ‘Sample rate x Bit depth x Number of channels‘. However, codecs typically use bitrate as a primary input parameter instead of those individual components. This is because bitrate directly controls the trade-off between audio quality and file size, making it easier to manage both encoding and playback requirements. It also abstracts away the internal implementation details, especially in lossy codecs like AAC or Opus, which use perceptual models and compression techniques that aren't strictly tied to sample rate or bit depth.

In this implementation, Opus and AAC were selected due to their broad support across platforms and good performance in web-based streaming scenarios.

Opus is a relatively newer codec, specifically designed for efficient, low-latency audio transmission over the internet. It has superior compression quality compared to other major codecs(including AAC)[**opusefficiency**], and often offers better space efficiency. Additionally, it is open and royalty-free, making it a good option for audio encoding. However, browser support is not universal—most notably, Safari offers limited support for Opus[**caniuseopus**], which restricts its use in HLS.

AAC, in contrast, is an older and more widely adopted codec, especially in the Apple ecosystem where it is natively supported across all devices and browsers. It is also a high-quality codec[**opusefficiency**] and still remains a standard in many streaming and media distribution systems. However, one very notable drawback of AAC is licensing – it is not royalty-free, meaning that commercial use, particularly involving encoding and distribution of AAC audio (as in streaming platforms), may require obtaining a license and paying fees to the patent holders via organizations like Via Licensing.[**vialicensing**]

Before selecting the encoding configuration, it is important to distinguish between two encoding strategies: **CBR (Constant Bitrate)**[**cbr**] and **VBR (Variable Bitrate)**[**vbr**].

CBR keeps the bitrate fixed across the entire audio file, resulting in predictable chunk sizes and smooth playback during streaming. VBR adjusts the bitrate depending on content complexity, which can provide better quality at a lower size compared to CBR but leads to inconsistent segment sizes, making streaming less stable [**cbrvbrcomp**].

For this reason, a **multi-CBR setup** was selected. While slightly less efficient in terms of quality-to-size ratio compared to VBR, CBR provides much greater reliability and predictability in adaptive streaming environments.

The following configuration, based on the official documentation of the respective codec implementations, was selected:

- **Opus**: 96, 160, and 256 kbps for DASH

- **AAC**: 96, 160, and 320 kbps for HLS
- **AAC-HEv2**: 24 kbps for both HLS and DASH

According to performance measurements provided by the codec developers, these bitrate values offer an optimal balance between audio quality and file size. Multiple bitrates are included to support adaptive streaming, allowing web players to dynamically select the most appropriate stream based on current conditions.

6.1.3 Processing

Audio processing is done via `ffmpeg`[**ffmpeg**] which is a tool that support both the encoding of audio and the manifest generation for both streaming protocols. It is written in C and supports multithreading ensuring high speed of audio conversion.

FFMPEG provides multiple codec implementations, out of which `libfdk_aac` and `libopus` were used. Again, the choices made were backed by the documentation details [**libfdkaac**, **libopus**].

In order to integrate it into the Backend architecture a small wrapper was written, which can be found in the `audio_processing/ffmpeg_wrapper.py` and `audio_processing/converters.py` files; a class diagram is also provided 6.4.

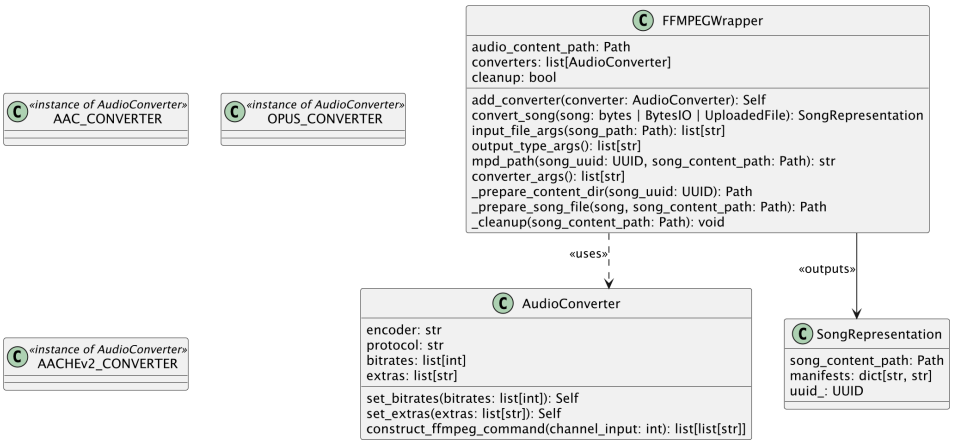


Figure 6.4: FFMPEG Wrapper

There are a couple of packages which already implement a wrapper around the library, however, they are not used because the needed functionality is minimal and introducing a big dependency is not necessary. In addition, the more popular and mature package `ffmpeg-python`[**ffmpegpython**] is not maintained anymore and e.g. OpenAI has dropped it as their dependency due to that reason[**ffmpegopenai**].

The resulting command for `ffmpeg` constructed by the wrapper(DASH only, for brevity) looks like this:

```
ffmpeg -i *input file path*
    -map 0:a -c:a:0 libfdk_aac -profile aac_he_v2 -b:a 24k
    -map 0:a -c:a:1 libopus -b:a 96k
    -map 0:a -c:a:2 libopus -b:a 160k
    -map 0:a -c:a:3 libopus -b:a 256k
    -f dash *output file path for manifest*
    -adaptation_sets id=0,streams=a
```

`-map 0:a` tells `ffmpeg` to select the audio track(s) from the first input.

`-c:a:*` specifies the codec for each output audio stream. The index (e.g., `:0`, `:1`, etc.) determines the order of the audio representations in the output DASH manifest.

`-adaptation_sets id=0,streams=a` tells ffmpeg to create a single adaptation set that will include all coded audio streams. That is needed for the ability to switch between them.

As the result, the individual manifests and corresponding chunks are created. Later on they will be placed under a subdirectory of Django's media directory, prepared to be served to the Frontend audio player.

Having determined, how the audio is going to be processed and represented, data and API model implementation is discussed in the further section.

6.2 Backend

The description is organized by the individual Django *apps*. Most of the subsections are comprised the following parts:

1. **Models** – Objects that define the data schema using Django's ORM.
2. **Serializers** – Code that converts Django models or native Python types into formats suitable for HTTP transmission (and vice versa).
3. **Views** – Handlers for HTTP requests, that provide the relevant business logic, and return appropriate responses.

As a means to simplify the understanding of the underlying architecture, a full class diagram can be seen first - it encompassed all attributes, methods and relationships of classes 6.5.

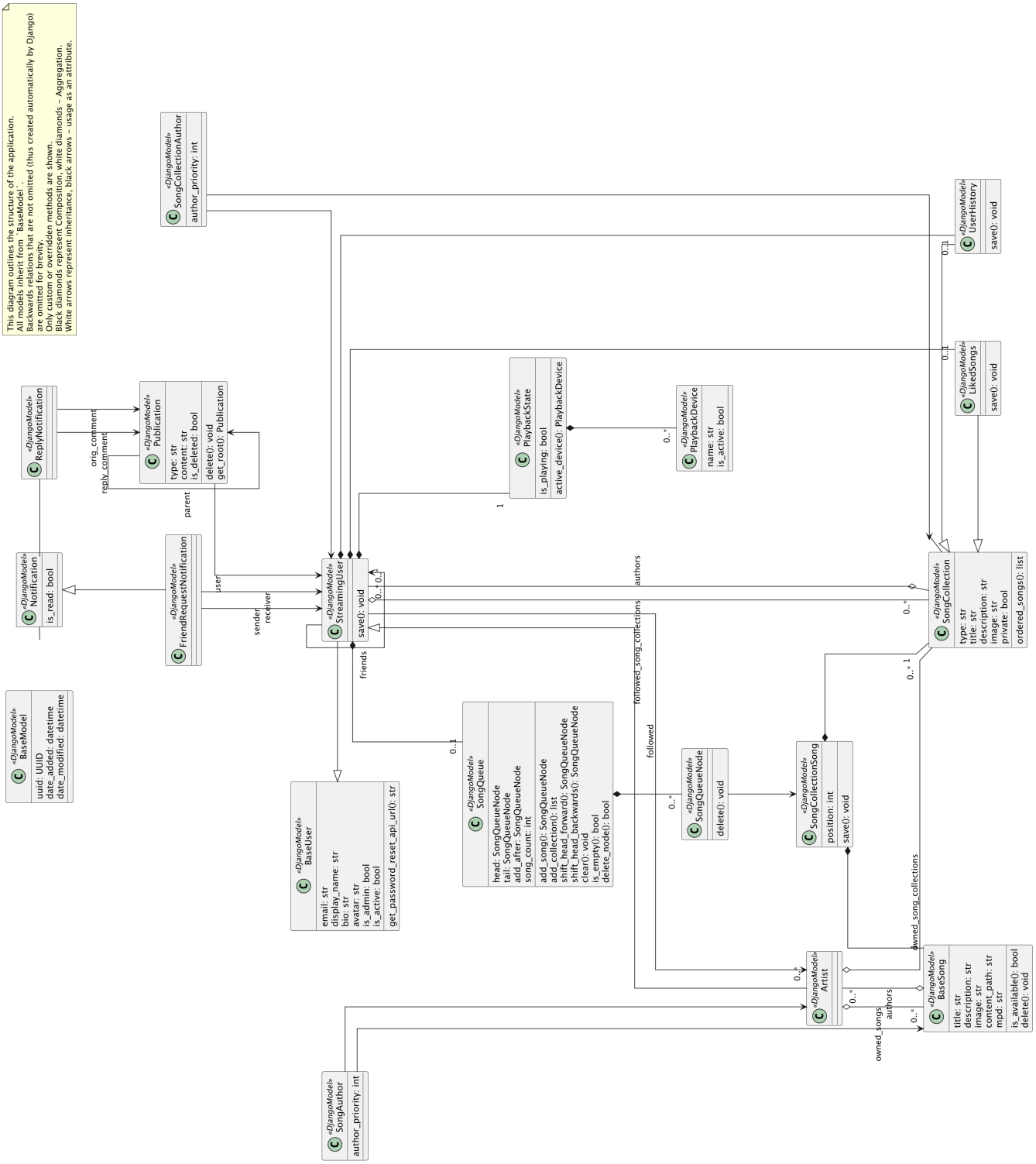


Figure 6.5: Backend Class Diagram

6.2.1 *base*

In this app the abstract `BaseModel` class is defined from which all other models inherit.

It has three attributes - `uuid, date_added, date_modified`. While the latter two are self-explanatory, `uuid` has to be explained. Since all of the data fetching in Frontend will be happening via the REST API, a unique resource identifier is needed. The drawback of using an `ID` of the model record, which is added by default for all models by Django, is the possibility of enumeration attacks **??** and unauthorized scraping of the web page. It is much easier to get an existing ID and simply send requests to API endpoints with an incremented ID value than to guess a random 128-bit string.

A retrieve serializer for the `BaseModel` is defined as well, it is also the base class for the serializers of other models.

6.2.2 *api*

Firstly, it is needed to say that the resulting API follows REST [**restdef**] practices; it only deviates from them when strictly following the rules would degrade the readability and usability of the API. This can happen, for example, when an endpoint is used to execute an action instead of a data operation. The other rule that is not followed is the presence of hypermedia links on related resources, as in the current architecture it makes more sense to use individual model identifiers and construct URLs from them instead.

The API itself is versioned, with current version being *v1*. In case of the following apps, all of API-related files are always defined under the `api/v1` folder.

As for the endpoint, all views that operate on data in some way require authentication, which can be determined by the presence of `permission_classes = [IsAuthenticated]` attribute on the view. User roles are also differentiated by checking the existence of the appropriate model, e.g. for Song creation the User set on request has to have a corresponding `Artist` instance.

6.2.3 *users*

User classes and token logic are defined in this app.

There are three main User classes:

- `BaseUser` - declares common fields for all other models, such as `email, display_name` and others.
- `StreamingUser` - the main class representing an account with privileges to use the streaming platform.
  Additional attributes are declared for it, namely:

  - Relations to other users: `friends`, which are other `StreamingUsers`, friend requests of whom the user has accepted (or the other way around);
    `followed` - the `Artists` to the updates of which the User has subscribed.
  - User-specific Song Collections: `history` containing all listened Songs,
    `liked_songs` containing favourited Songs,
    and `followed_song_collections` - the Collections which user has saved to their profile.
  - `song_queue` - a special structure containing already/to be listened to Songs of the User,
    it will be described later in the dedicated app.

- `Artist`: a subclass of `StreamingUser` with additional capabilities, such as Song uploads.

JWTTokens are also customized in this app. They are based on the implementation from `djangorestframework-simplejwt` [**simplejwt**]. The token system follows the standard logic of access and refresh tokens: the access token is short-lived and used to authenticate regular API requests, while the refresh token is long-lived and can be used to obtain new access tokens without requiring the user to re-authenticate.

Authentication for each HTTP request involves attaching the access token as an HTTP Bearer header. After parsing the request, token is retrieved and validated. After validation, the User identifier is extracted and used to identify the corresponding User record. User validation logic is changed so it uses User's `uuid`, hence it is added to the token payload. This is done for the same reasons that were outlined in the *base*.

Basic serializers for all User models are declared, allowing for the retrieval of information about specific Users or their creation and updates.

The following views are added as well:

- `UserRetrieveUpdateView`
- `UserCreateView`
- `UserFriendsView`
- `UserFollowedView`
- `UserFriendsCreateDeleteView`
- `ArtistFollowersView`
- `UserEventViewSet`

The purpose of all views, except for the last one, is to send the information about Users or perform create/update actions on the related data.

UserEventViewSet is a special view used to implement the SSE connection, the details are provided in the subsection *sse*.

Views for the token generation and renewal are also declared. Their respective endpoints are:

- `/api/v1/token/` — accepts user credentials (email and password) and returns a token pair (access and refresh tokens).
- `/api/v1/token/refresh/` — accepts a refresh token and returns a new access token.

6.2.4 *sse*

As was mentioned in Implementation Planning, `django-eventstream` package is used for the Server Sent Events configuration. Internally it works by creating special objects called *channels* to which clients can subscribe via an HTTP request. Then, when a new event for a specific channel is created, e.g., via the `django-eventstream.send_event()` method, all clients subscribed to that channel will receive an HTTP response with the specified content.

'UserEventViewSet' listens for incoming SSE connection requests from Users and creates a personalized channel using User's UUID.

Events are used throughout the system mostly to signify to the Frontend that some data is stale. An example of this is the aforementioned 'UserFriendsView', the post request handler of which is defined like this:

```
class UserFriendsCreateDeleteView(APIView):
...
def post(self, request, *args, **kwargs):
    ...
    user.friends.add(sender)
    send_invalidate_event(
        EventChannels.user_events(user.uuid),
        ["user", "friends", str(user.uuid)]
    )
    ...
```

Here, the custom `invalidate_event` signifies to the Frontend that the `friends` list has changed and should be refetched.

The *invalidate* logic is present for views that operate on models that are used in the Frontend, specifically in view handlers which change data. That way the client always has fresh information.

Event re-delivery is also handled by the package set with this parameter in Django settings:

```
EVENTSTREAM_STORAGE_CLASS = \
    'django_eventstream.storage.DjangoModelStorage'
```

This tells the package to store the events in the database, and in case of network failure or client disconnect they are redelivered.

Since the SSE part has been explained, it is safe to consider the main part of the application – the *streaming* package.

6.2.5 *streaming*

This app defines the main models responsible for the audio content and playback handling.

Songs and Song Collections

The backbone of the application are the `BaseSong` and `SongCollection` models which are declared in `streaming/songs.py` and `streaming/song_collections.py` files.

- `BaseSong` contains the metadata about each individual song - `title`, `description` and `image`.

  Then there are attributes which contain the information about the underlying song file: - `content_path` representing the root directory with Song files(chunks, manifests) - `mpd, m3u8` providing manifest paths for DASH and HLS respectively.

  The last attribute is `draft`. It used in the Song creation process, which will be explained in the view description.

- `SongCollection` represents a container for Songs and can be of two types(set with `type` attribute) – either a Playlist or an Album. Playlist is a collection that is formed out of existing Songs, which can be created by both StreamingUsers and Artists. An Album, on the other hand, is a collection that can be created only by an Artist containing unique, uploaded Songs.

  Because an individual `BaseSong` can be in multiple collections, a Many-to-Many relation is defined between `BaseSong` and `SongCollection` named `SongCollectionSong`. To enable arbitrary ordering of the songs a `position` attribute is added.

  Another attribute on the `SongCollection` is `comments`. As the name suggests, these are pieces of text that each User can add to the collection. The underlying model is discussed in the social.

The same metadata as in `BaseSong` can be set as well. However, there is one extra attribute - `private`. It is used to signify which collections should not appear e.g., in global search results.

In addition, there are two subclasses of `SongCollection` - `UserHistory` and `LikedSongs`. They are moved to separate classes, because in the future they will most probably have custom logic added to them. In the current version of the app only the ordering that makes the newest added songs appear first is defined.

Each `SongCollection` and `BaseSong` also has a relation to the Users that created it(and visa versa). It is defined as an M2M relationship with an intermediary table - `SongCollectionAuthor` and `SongAuthor`. Attribute `author_priority` sets the order in which the authors should be returned from the API and displayed on the Frontend.

As for the views and endpoints - a standard set of CRUD operations is implemented for both models. Additionally, there are special views for User interactions with models, such as `SongAddLikedView`, which adds the provided Song to User's `LikedSongs` or `SongCollectionRemoveSong` which lets the User remove a Song from a collection, if it was created by that User.

However, a couple of things have to be said about Song and Collection creation. The design choice is to not allow users create isolated Song instances, instead, if the User wants to upload Songs separately from a Collection, a single-song Album would be created instead. This greatly simplifies the logic when handling User playback interactions later on. For example, it makes possible to use the aforementioned `SongCollectionSong` instead of `BaseSong` in most views. As this relation contains the information about both models, we can, for instance, accurately show not only which Song is playing, but a Collection as well, without tracking both models. There are further different benefits of this which could be seen in the Frontend implementation.

Apart from that, the Collection creation process is multi-stage and two views are responsible for it: `SongCreateView` and `SongCollectionCreateView`. Whenever a User wants to create a new Collection, firstly the Songs are processed. If an error happens during this stage the response with successful and failed Songs is returned and `draft` attribute is set to *True* for the created Songs. This is done, because the User can abort the process of creation e.g., by closing the whole browser application, and there would be no proper way to signal what Songs should be deleted. A simple crontab job or `celery` library could be utilized to run periodic tasks which would clean up Songs in the draft state.

When creating a Song, multiple validation stages are made for the User input, both on the Frontend and the Backend. For instance, `ffmpeg_wrapper.py` checks the audio file size, its mime type via [**pythonmagic**] library and the underlying format, with available formats specified in `ffmpeg_wrapper.`

`ALLOWED_FILE_TYPES` variable.

Having discussed the main building blocks of the audio content part of the application, `SongQueue` which operates on these models can be presented.

SongQueue

`SongQueue` represents a list of previously listened Songs and the ones that are going to be listened by the User.

The choice of the underlying structure was made based on the desired operations. Basic addition and removal from the queue had to be supported, as well as the differentiation between playing a Song/Collection or adding it to the queue. For example, when user would click play on a Song, it had to become the new "head" of the queue. And in case of clicking "add to queue" this Song would be added after other items that have been queued previously. All of this implies that addition/removal can happen in any part of the list. As the result a Doubly Linked List pattern has been chosen for this model.

As the queue information had to be persisted in the database, a common implementation with nodes that point(using foreign keys in that case) to next and previous node might have been not the most efficient. An intersting approach proposed online was to use a *position* column instead of foreign key references. That way we would be able to retrieve the whole queue with an SQL statement like so:

```
SELECT *
FROM (join node table, queue table and StreamingUser table
      to find the queue nodes of a particular User)
WHERE node.position > (join queue and nodes to find the head of the queue)
ORDER BY node.position
```

While using a `position` column simplifies queue traversal and ordering, it introduces potential issues when it comes to frequent inserts and deletes.

Every time a node is inserted between two existing positions, a new `position` value must be chosen such that the ordering remains correct. If positions are represented by integers, repeated insertions between tightly packed values (e.g., inserting between position 1 and 2) will eventually require re-indexing of the entire list to free up space — an operation that can be expensive in terms of performance. Similar problems can occur during deletes as well.

These limitations could be mitigated, for example, by using high-precision floating point numbers, powers of numbers or large increments like 1000 to delay the need for full re-indexing. However, without the data showing the

expected usage of the queue it is hard to tell, whether this approach would be beneficial.

That is why a more simple approach with identifiers of neighbouring nodes was chosen, at least for now.

Two models make up the queue functionality - `SongQueueNode` and `SongQueue` itself.

Each node carries information about the `SongCollectionSong` attached to it, as well as links to the previous and next node.

`SongQueue` has foreign keys to the nodes which signify the `head` and `tail` of the queue. Moreover, a special, third, attribute is declared - `add_after`. This is also a foreign key to one of the nodes, but this time it indicates the node after which items should be added, when "add to queue" action is selected. I.e. if the user had selected "play" on a Song, it would become the queue head. Then, if the user selected "add to queue" on five different Songs, then the head would still point to the playing Song, but `add_after` would contain the key of the fifth Song added using the second action.

Multiple operations are supported, with the main ones being:

- Setting the queue head. In case of Collections, e.g., when the User clicks "play" on a Collection object, the entire Collection is added to the queue.
- Shifting the queue head backward and forward.
- Removing a node.
- Clearing the entire queue.

The final aspect that needs to be addressed is the managing of the playback state.

Playback State and Playback Devices

The term "playback" represents the state of the sound, whether it is being played in a given moment. Each User has an attribute named `playback_state` represented by the `PlaybackState` model. The playback can be toggled by using the `PlaybackState.is_playing` attribute.

Because the User can open multiple instances of an application, the playback handling can not be unloaded to the Frontend only, as it would not be synchronized across the instances of the application.

As the result, the application has to differentiate User's "sessions", which is done via the `PlaybackDevice` model. It represents the individual devices on which the application is currently open. Each device has `is_active` attribute, signifying where the sound should be played, and where to display controls(seek bar, volume).

It would be expected for all instances of the application to show the same playback state(playing/stopped), and, intuitively clicking the play button would have to result for playback to be toggled on the active device with visual confirmation on all other devices. In addition, the active device must be able to be switched, which is done by the appropriate `PlaybackDeviceActivateView` handler.

The standard serializers and views that allow for state/device manipulation are implemented. However, `PlaybackDeviceDeleteView` deviates from the standard rules, as it allows not authenticated requests and softens the CSRF restrictions. This is done due to browser limitations, shown in the **??**. The authentication is still made "by hand", inside the handler itself.

This concludes the audio part of the application.

6.2.6 social

Before examining the contents of the social app, an important Django concept has to be brought up – the `ContentType` framework and `GenericRelation` [**djangocontenttype**].

The `ContentType` framework provides a way to refer to any model in the system using a generic identifier. `GenericRelation` uses the `ContentType` framework by storing the target model's type and primary key in two separate fields: a foreign key to the `ContentType` model and an object ID. Together, these allow a single relation to point to any instance of any model, enabling polymorphic associations without hardcoding specific model references.

This concept is used to implement the `Publication` model.

For better understanding, the model code can be found below. Here, the `content_object` field is a `GenericForeignKey` that dynamically joins the `content_type` and `object_id` fields to establish a reference to any model instance. This allows each `Publication` instance to be associated with an arbitrary object, such as a Song, or a Collection without requiring a fixed foreign key to a specific model.

```
class Publication(BaseModel):
    class PublicationType(models.TextChoices):
        COMMENT = "comment", "Comment"
        POST = "post", "Post"

    RELATED_MODEL_TYPE_MAP = {
        "user": "users.StreamingUser",
        "collection": "streaming.SongCollection",
        "song": "streaming.SongCollectionSong",
    }

    type = models.CharField(choices=PublicationType.choices)
```

```
    content = models.CharField(max_length=255)
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL, null=True,
        on_delete=models.SET_NULL
    )
    parent = models.ForeignKey(
        "self", null=True, blank=True,
        related_name="replies", on_delete=models.SET_NULL
    )
    is_deleted = models.BooleanField(default=False)

    content_type = models.ForeignKey(
        ContentType, null=True,
        blank=True, on_delete=on_content_type_delete
    )
    object_id = models.PositiveIntegerField(null=True, blank=True)
    content_object = GenericForeignKey()
```

Another attribute that needs an explanation is `type`. Currently it can have one of two values - `comment` and `post`. The comment type is used for comments under audio content(collections), while `posts` represent standalone User publications.

The `Publication` model also contains a `parent` attribute, allowing for building trees of publications(i.e., replies).

Standard CRUD serializers and views for publications are provided, with the addition of custom handling of the related models.

Views related to the publication creation also have a special handling of instances which are considered as replies. They create a special `ReplyNotification` object, discussed in the next subsection notifications.

6.2.7 notifications

In order to notify users about actions such as replies to their publications or new friend requests, a notification system was implemented.

The following two notification types are currently supported:

- `ReplyNotification` – created when a user replies to a publication (e.g., comment). It stores references to both the original and reply comments. The corresponding manager triggers an SSE event to the original author's channel, making Frontend refresh the notifications list.
- `FriendRequestNotification` – created when a friend request is sent. It stores references to both the sender and the receiver. Upon creation, same as for `ReplyNotification` an SSE event is emitted to the receiver's channel.

Both classes inherit from `Notification` class. It has a single attribute – `is_read` which helps to differentiate between already seen and new notifications on the Frontend

6.2.8 *recommendations*

The *recommendations* app currently provides basic search functionality across Songs, Collections, and Users using Django's native PostgreSQL full-text support.

The core search logic is implemented in the `SearchView`, which uses `TrigramSimilarity` to retrieve fuzzy matches based on string similarity. Trigram search is appropriate for this purpose because many of the search targets, such as usernames or collection title, often contain informal, abbreviated, or non-standard text. In such cases, traditional vector-based full-text search (e.g., via `SearchVector`) would perform worse, as it is optimized for longer, more "text-like" content.

A dedicated search engine like Elasticsearch was not introduced for two reasons:

1. The scale of the platform does not currently justify the additional infrastructure, deployment, and synchronization complexity that comes with search engines.

2. PostgreSQL's trigram extension provides sufficient performance and accuracy for the expected use-case. Moreover, Django offers first-class support for it through the `django.contrib.postgres.search` module.

This design ensures efficient search behavior while avoiding unnecessary overheads.

With the API and server-side logic in place, the focus now shifts to the Frontend, where user-facing interactions, state management, and playback handling are implemented.

6.3 Frontend

As stated in Implementation Planning, React was used as the library for the Frontend implementation. Alongside React these tools were used as well:

- `shadcn/ui` [**shadcn**] and `tailwindcss` — used to build UI components.
- `@tanstack/react-query` [**tanstack**] manages data fetching, caching, and synchronization; `axios` is used for making HTTP requests.
- `react-router-dom` [**reactrouterdom**] — handles client-side routing and navigation between views.

- `react-hook-form` [**reacthookform**] — provides form state management.
- `Vite` [**vite**] — used as the build tool for development and production.

The development cycle mostly looked like this:

1. Define a static component with pure JSX, outline what data it needs and where it should be placed.
2. Define the state logic inside the component.
3. Add API endpoints and queries/mutations needed by the component.
4. Test the component with actual data from the Backend.
5. Refactor, e.g., lift state, move data to a provider, or split a larger component into smaller ones.
6. Style the component using `shadcn` and `tailwindcss`.

6.3.1 Authentication

Since most of the API endpoints require authentication a token has to be added to every request. Firstly, on User log in, the applictation tries to obtain a token pair by sending a plain POST request to the appropriate endpoint. User credentials are added to the payload, but as the application is served only with HTTPS, and the payload is encoded, everything is secure.

After getting the tokens, they are added to the Cookie storage with a specified expiry time. It is worth to mention, that this is not the best practice when working with tokens; storing tokens in JavaScript-set cookies exposes them to **Cross-Site Scripting** attacks.

A more secure approach is to store the access token in memory(e.g., as a JS variable) and the refresh token in an HttpOnly, secure cookie. However, the Backend would have to be heavily customized for that, and the implementation is left for the future versions of the application.

After getting the tokens, every further request needs to set them as the HTTP *Bearer* header. This is done via a feature called axios interceptors [**axiosintercept**]. They allow to add any custom logic to be executed before or after the request.

The request interceptor checks, whether the access token exists, and if it does, adds it to the request. The response interceptor has an "on error" handler for responses with NOT AUTHORIZED status code(401), this usually means that the token was not set on the request, hence, it tries to obtain a new one by sending request to the token refresh endpoint. If the token is successfuly refreshed, the original request is retried. Otherwise the request is rejected.

The implementation of the discussed techniques can be found in `src/auth/authentication` and `src/config/axiosConf`.

Next the actual interactions with API and data management are presented.

6.3.2 Data Management

Data management is performed with `@tanstack/react-query` library, especially with its `useQuery` and `useMutation` hooks. These hooks handle asynchronous data fetching and mutation while providing built-in caching, automatic retries, and background updates.

An example of a query and mutation can be seen below:

```
    export function usePlaybackRetrieveQuery() {
    return useQuery<IPlaybackState>({
        queryFn: async () => {
            const res = await api.get(PlaybackURLs.retrieve);
            return res.data;
        },
        queryKey: ["playback"],
    });
}

    ...

    export async function commentCreate({ objType, objUUID,
        content, replyToUUID }: IAddCommentParams) {
    const data = {
        content,
        parent: replyToUUID,
    };
    await api.post(CommentURLs.commentCreate(
        objType, objUUID
    ), data);
}
```

An important note is the `queryKey` attribute in the `useQuery` hook , which is used to label and organize the data being fetched. A query can be "invalidated" using that key, which means that React Query will mark the data as outdated. Then it will automatically fetch fresh data the next time it's needed -— like when the component shows again or the user focuses the tab.

Some data, which has to be accessible on multiple levels of the application(in different components) are moved to providers. They enable access to data, without the need of refetching in each component. Moreover, each

component which depends on the data from the provider will be updated when this data changes.

An example of a provider:

```
    export function UserCollectionsProvider(
        { children }: UserCollectionsProviderProps
    ) {
    const userUUID = useUserUUID();

    const { data } = useQuery({
        queryKey: ["collectionsPersonal"],
        queryFn: userUUID ? () =>
            fetchCollectionsPersonal(userUUID) : undefined,
        enabled: !!userUUID,
    });
    const contextValue = {
        history: data?.history ?? null,
        liked_songs: data?.liked_songs ?? null,
        followed_collections: data?.followed_collections ?? null,
    };

    return (
        <UserCollectionsContext.Provider value={contextValue}>
        {children}
        </UserCollectionsContext.Provider>
    );
}
```

And the usage of the context it provides:

```
export function LeftColumn() {
    const { liked_songs, history, followed_collections } =
        useContext(UserCollectionsContext);
    //render the collections
}
```

These techniques alongside the React state management make data management efficient, ensuring proper caching and reusing of data.

6.3.3 SSE

The Frontend implementation of Server Sent Events is provided in the `src/hooks/useEvent` hook. It leverages the `eventsource` library [**eventsource**] to manage the low-level SSE connection, while the hook itself implements custom logic for authentication, reconnection, and message handling.

As was presented in *sse*, the only type of messages implemented right now is an "invalidate" message. This message contains one parameter -

the key of the query to be invalidated. That way we can control from the backend what data and when to refresh on the frontend.

The connection is created only once, in the `HomePage` component.

6.3.4 Playback and Player

As both *DASH* and *HLS* deliver segmented media via manifest files, a compatible player is needed on the frontend to handle parsing, buffering, and adaptive streaming.

`Shaka Player` [**shaka**] was chosen for this purpose, as it supports both DASH and HLS with a unified API. Moreover, it integrates well with React components, and provides features like adaptive bitrate switching and error handling.

An alternative approach would be combining the MPEG-DASH reference player [**dashref**] with `hls.js` **??**,but this would require handling two separate libraries with different APIs and event models, making the playback logic more complex. Shaka Player simplifies this with a single consistent interface.

Then it is needed to differentiate between browsers in order to select the appropriate manifest to download. Two simple functions are defined to get the proper manifest path:

```
    const isSafari = useMemo(() => {
        return /^((?!chrome|android).)*safari/i.test(navigator.userAgent);
    }, []);

    const url = useMemo(() => {
        if (!playingCollectionSong) return undefined;
        return isSafari ? playingCollectionSong.song.m3u8 : playingCollectionSong.song.m
    }, [queueHead, playingCollectionSong, isSafari]);
```

The playback itself is driven by different conditions. For example a User can click a `SongPlayButton` and that will either stop or resume the playback or start a new one with the specified Song. One of another such conditions is the end of the playback of the current Song, in such case the mutation responsible for the queue head shift is invoked and the playback is either initialized with the next Song or stopped, if the queue is empty.

6.3.5 Device Handling

User device management wast one of the most challenging parts.

The expected behaviour from the application would be to register a new device, when the User logs in, then show the list of devices on which the service is running, and when a new device is chosen, switch the playback to

it. When the application is closed the device should be removed from the list of active devices.

The registration and switching part is not complicated – it is possible to send a device registration request on the first render of the `App` component, which is mounted only once. The result of this request, containing the unique identifier of the device then would be added to the LocalStorage. Then, on page refresh, the presence of this variable could be checked and additional requests omitted.

However, issues arise when the device needs to be removed. Modern browsers do not provide a reliable mechanism to trigger actions on page or window close [**chromelfapi**, **beacons**]. The `beforeunload` and `unload` events, which signal page refresh, redirection, or closure, are inconsistent – particularly on mobile platforms and Safari.

Chrome documentation recommends using the `visibilitychange` event to detect when a page becomes hidden. While this event is more consistently supported, it is triggered by actions such as tab switches or window minimization, which do not imply that the playback device should be deactivated or removed.

Even implementing heartbeats – periodic requests to the backend indicating that the device is still in use -— would require additional backend logic to manage device lifecycles. A more robust solution could be an external process, for example, based on the `celery` library, to monitor and control device lifespan. However, this approach remains to be implemented.

For now the handler works with the `unload` effect, as it is proven to fire in around 90% cases on modern desktop browsers [**beacons**].

6.3.6 Components

All UI components follow a consistent structure in both composition and logic.

The component hierarchy is decomposed into several levels:

- **Groupings and Logic Providers** – High-level wrappers that provide context, routing or shared state to nested components, for example `App` or `HomePage`.
- **Layout Components** – Define page structure and positioning (e.g., `Header`, `RightColumn`, `PlayerBox`).
- **Data Representation Components** – Render fetched data and define interactions on it.
- **UI Subcomponents** – Reusable elements like buttons, inputs, and fields used across higher-level components.

This layered organization ensures clear separation of concerns, simplifies maintenance, and promotes reusability.

The use of the `shadcn/ui` component library and `lucide-react` icon set further improves consistency and maintainability across the interface by providing standardized, customizable, and accessible UI building blocks.

Once both Backend and Frontend were fully developed and tested, the final step was to make the platform accessible to users.

6.4 Deployment

In order to make the application publicly accessible, it was deployed to a virtual private server hosted on Hetzner. The project uses Docker to build and isolate the application environment, including containers for the Django backend, PostgreSQL database, and static file serving.

Separate Dockerfiles are provided for the Backend and Reverse Proxy + Frontend build. The both builds are split into multiple stages to ensure the smallest image size possible.

One thing has to be pointed out about the Backend image. `ffmpeg`, which is used for audio processing, is not installed as a package or a binary, but is instead built from source. This is due to the restriction of the `libfdk_aac` [**libfdkaac**] library, which is not free software and cannot be distributed in precompiled binaries due to licensing constraints. As a result, the library must be compiled manually and linked during the build process.

Images then are built locally and pushed to a private Docker registry for reuse and ease of production deployment. Docker compose is used for the orchestration.

The web server and reverse proxy are handled by `Caddy` [**caddy**], which simplifies HTTPS setup and certificate renewal. Domain configuration was managed through `Porkbun` [**porkbun**], which provided DNS records pointing to the Hetzner [**hetzner**] server.

The application itself can be accessed at `https://musikk.stream`.

Chapter 7

Summary and Conclusion

This thesis set out to design and implement a modern music streaming platform with integrated social features, addressing the observed gap in user interaction capabilities across existing services. Based on the outcomes of a conducted survey and analysis of current solutions, the project identified an existing user interest in socially-enhanced music consumption and sharing.

The platform was implemented using a Django-based backend and a React-based frontend, chosen for their development efficiency, community support, and extensibility. The audio delivery is performed using adaptive streaming via DASH and HLS protocols, ensuring compatibility and quality across devices and browsers. Opus and AAC codecs were utilized with multiple CBR representations to support adaptive bitrate switching. Server-Sent Events (SSE) were implemented to provide real-time updates with low complexity and good reliability. In addition, the resulting application is synchronized across different instances.

The resulting system supports all basic operations expected from a music streaming platform, while adding additional social features. The social interaction capabilities were designed to be lightweight yet functional, including post and comment systems, real-time friend activity updates.

Deployment was carried out using Docker, PostgreSQL, and a Caddy server, and the application is now publicly available online. Key challenges, such as browser inconsistencies with playback state persistence and device lifecycle handling, were identified, and workarounds or future improvement paths were proposed.

In conclusion, the project demonstrates that a socially oriented music streaming platform can be effectively implemented with modern web technologies. It fulfills the identified requirements, introduces meaningful social interactions, and presents a flexible foundation for further enhancements, such as collaborative playlists, advanced discovery algorithms, or integration with third-party media platforms. Future development will aim to expand these features, improve performance, and ensure long-term scalability.

Sources