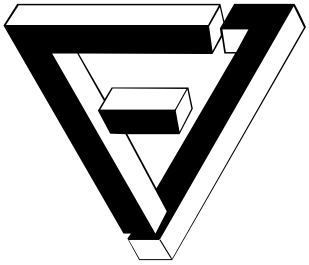
Masarykova univerzita
fakulta informatiky



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

2.1 Background and objectives

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 examines the individual content-consumption preferences, listening and discovery habits, platform usage patterns, and social behaviours.

2.2 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 consists 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. Ungrouped answers could be found in the original survey.

As for the respondents - 119 people had participated, with most being from Russian-speaking countries; the majority was in the 18 to 30 age group, with the exact distribution shown in Figure 2.1.

The form can be accessed at: https://docs.google.com/forms/d/1vhhAu_SfuHV4xy6JoDaRsM5uCElYn_csTmN8u4ZlpHc

2.3 Results

2.3.1 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 dedicated platform with advanced features might not be relevant for most

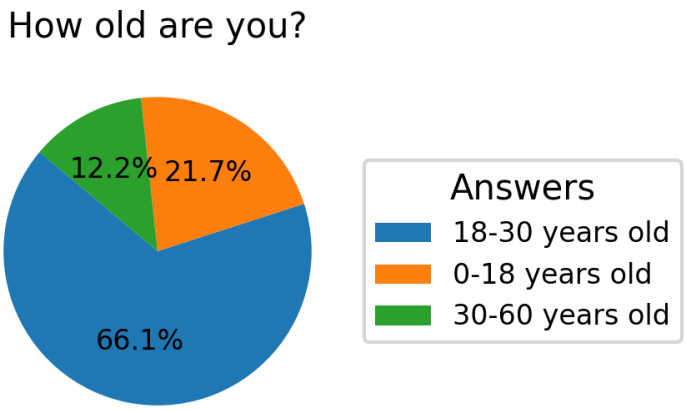


Figure 2.1: Age distribution of respondents

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 would be to discover how the audio content is mostly accessed.

2.3.2 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.

Regarding the platforms themselves - Spotify has taken the first place in terms of popularity, proving the global statistics[**spotifypopularity**]. However, Yandex Music being in the second place deserves an explanation. As mentioned previously, most of the respondents are from Russian-speaking countries, Russia specifically. With a lot of western companies leaving its market in 2022, music streaming services included, most of the users has 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 or JioSaavn for the Indian one, as all of the respondents were based in the european part of the world. The full statistics are shown in Figure 2.5.

Being established that streaming services are indeed widely-used and are in demand, the next important step would be to analyze the regularity of

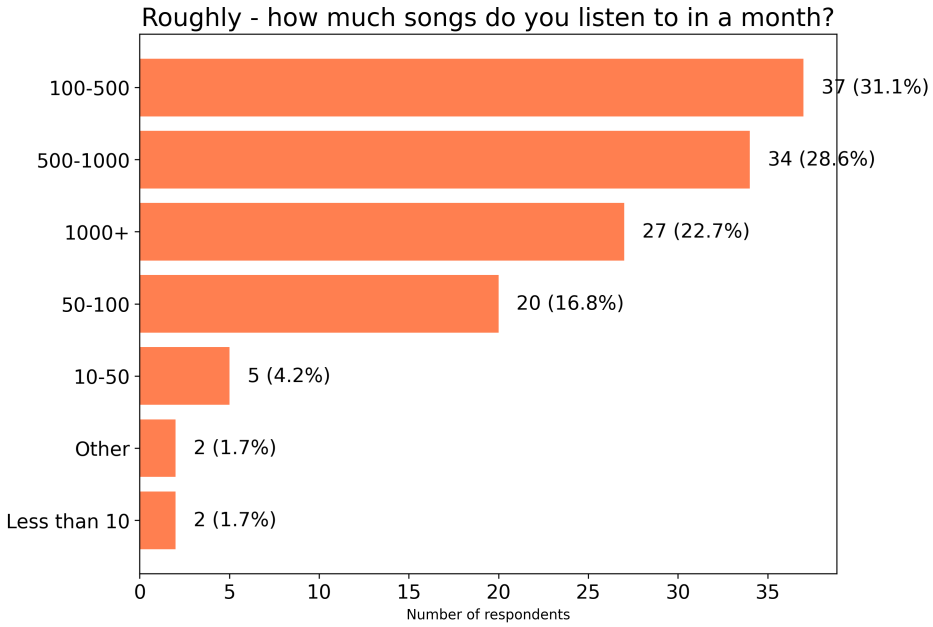


Figure 2.2: Monthly song count per respondent

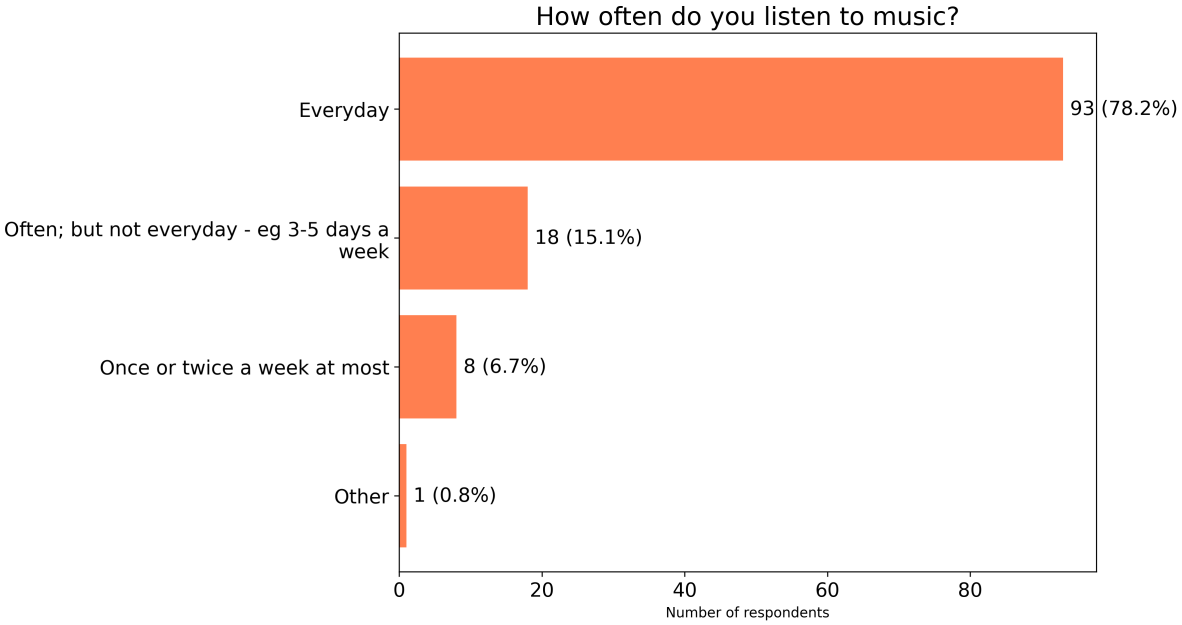


Figure 2.3: Listening frequency of respondents

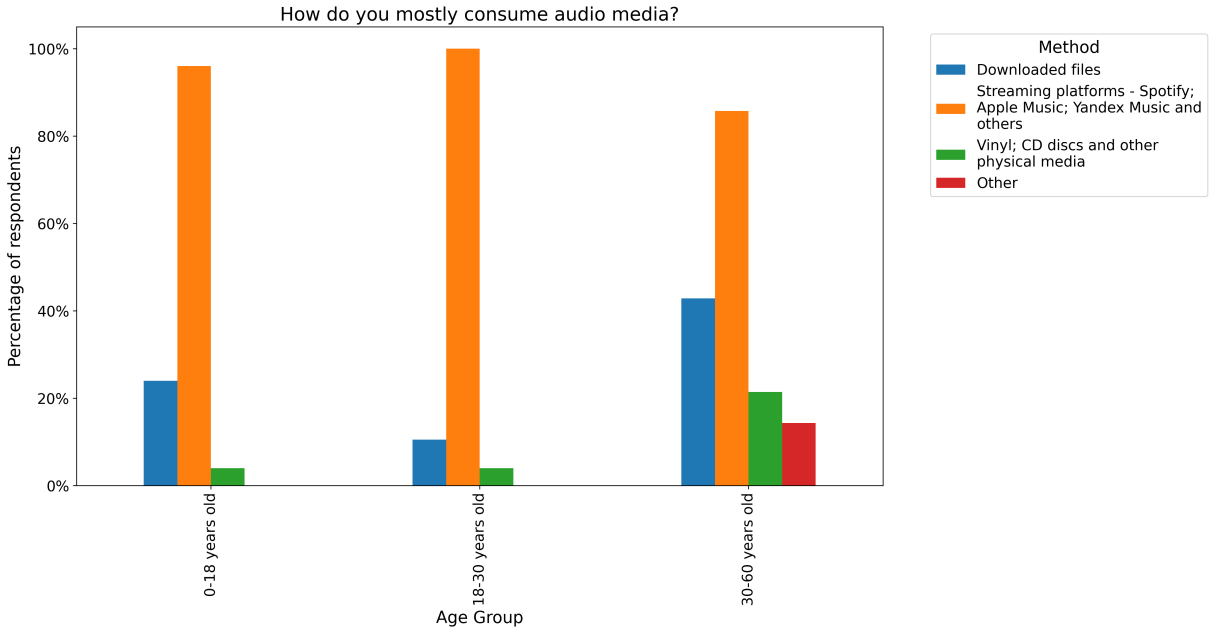


Figure 2.4: Audio media consumption by age group

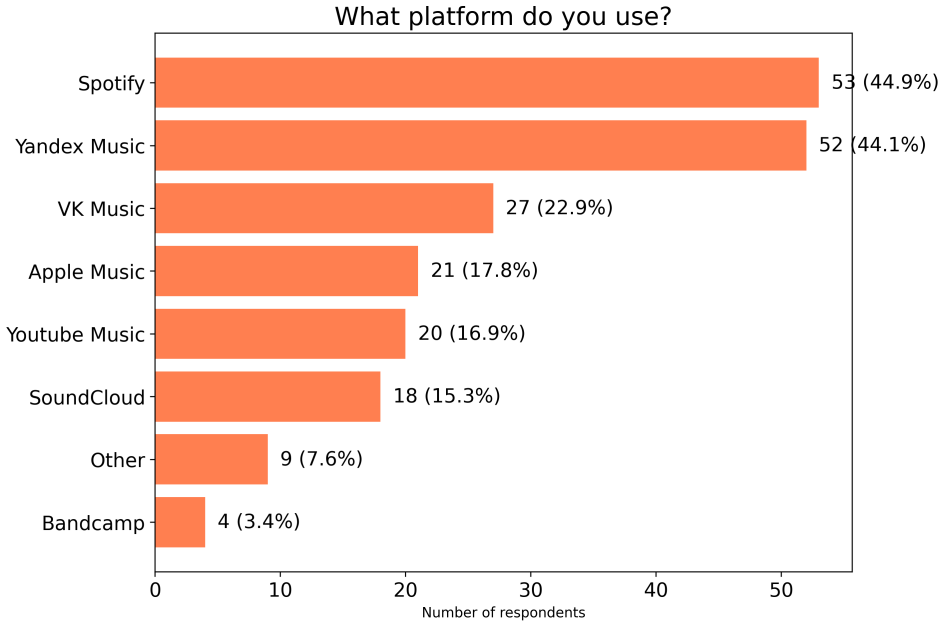


Figure 2.5: Streaming platforms popularity

music-centered social interactions.

2.3.3 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, though not as prevalently as in-person scenarios.

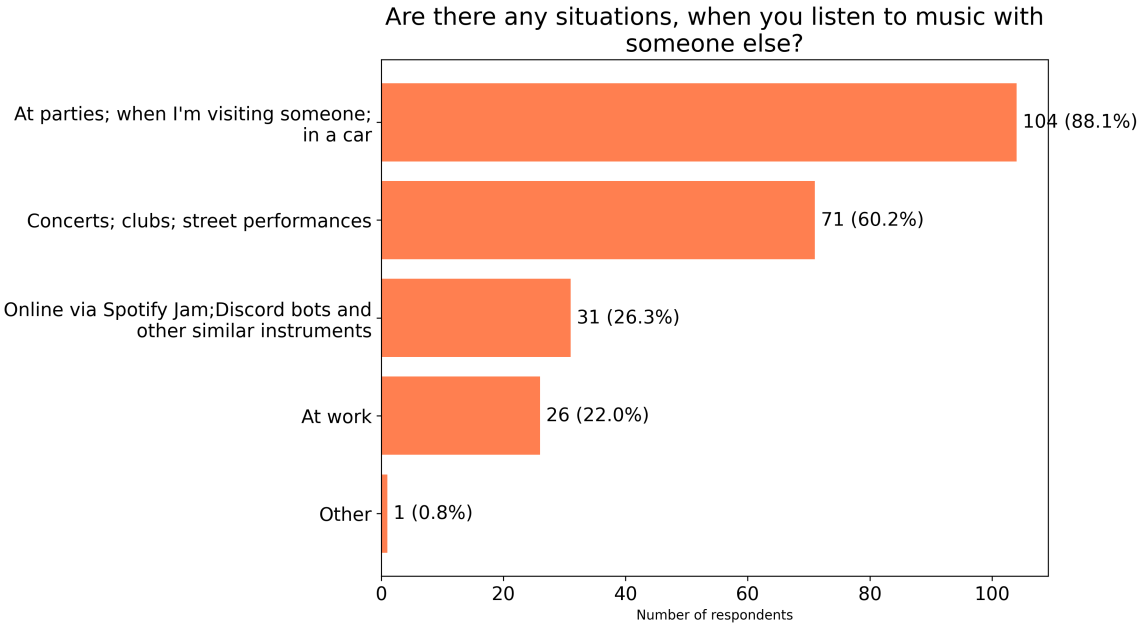


Figure 2.6: Social interaction methods

Question 11 (Figure 2.7) investigates how often these interactions occur — the 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 following - "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: a majority of participants reported engaging in these exchanges regularly. Over 70% (Figure 2.8) indicated that they share music at least

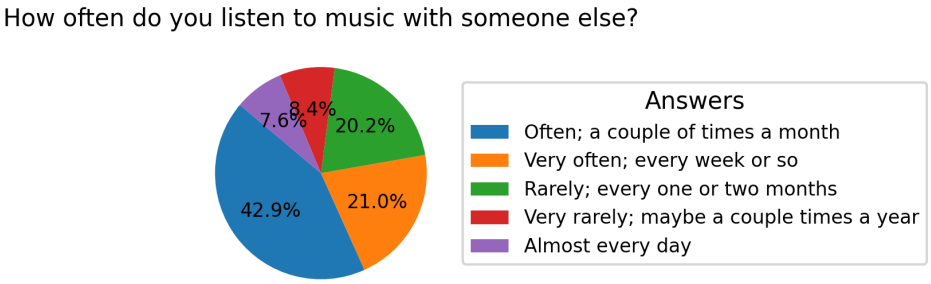


Figure 2.7: Social interactions frequency

several times a month, with many doing so weekly or more often. This further emphasizes the active social role of music.

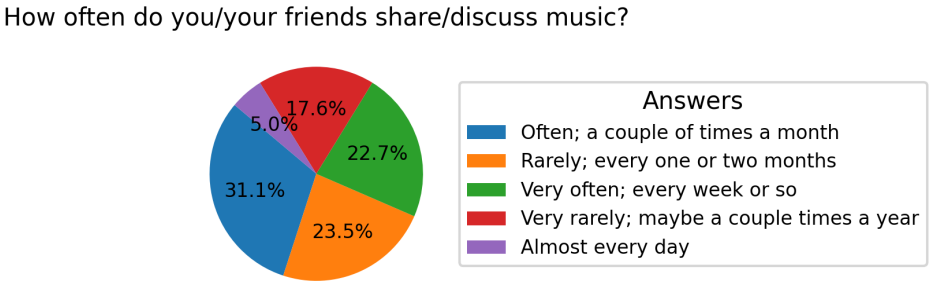


Figure 2.8: Music sharing frequency

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 a clear 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 streaming platform. This is further supported by the responses to Question

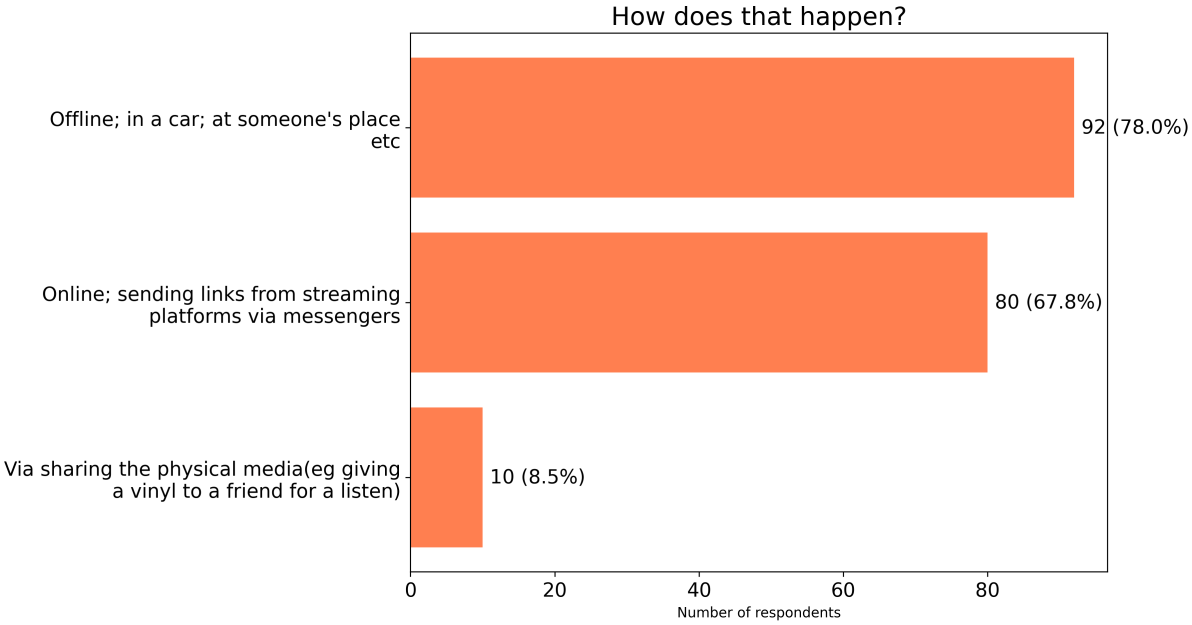


Figure 2.9: Music sharing methods

15, where over 60% of participants stated they would use additional social features, if available on a streaming platform.

2.4 Summary

Overall, the results show that music is a vital part of many peoples' 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 the possibility for interactions with other users. Therefore, the idea of a music streaming service with integrated social features could be relevant, addressing real user needs.

Chapter 3

Existing Platforms

As a way to better understand what instruments people use when interacting with music, it is useful to examine existing solutions. Additionally, this research is going to provide an 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 also a popular way to interact with music as was shown in ...

3.1 Streaming Services

As can be seen in the table 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 the streaming services. There are many options available, but only those that are both popular and have unique elements will be considered in this thesis. The descriptions, rather than offering 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 which 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 audio pref-

erences 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 which 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 4 tabs: 'collection', 'wishlist', 'followers' and 'following'. By far the most interesting feature is under the 'following' tab – users can subscribe to available genres and then specify the ones that 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 is able to 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.

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.

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

3.2.1 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 active to this day. This is a country-specific resource, and as a consequence of that, on top of providing the regular music and forum components, there are elements which are 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

Another memorable outcome of the survey is the high percentage of music discovery on non-music specific platforms. Instagram, YouTube, TikTok and other services that are able to host user-created content can have a major effect on listening habits. In recent years, music has been more deeply integrated into these platforms. For instance, Instagram introduced a dedicated audio tool to seamlessly embed sounds and songs into 'Reels'[**inst_audio**], while YouTube automatically detects songs used in videos and adds them to the video description.

Summary

As it can be clearly 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 the potential for social interaction among users.

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 have the biggest 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 main language for this part of the application. It has support for 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 time of the development, it was decided that a framework would be used.

5.1.1 Framework Comparison

Currently, there are 3 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 a sense that almost everything - from access control and database management to API routing and 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 of the notable drawbacks include its opinionated design choices, due to which it can be 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 a lot of 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 in selecting 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 more 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. For other frameworks a separate library would have been needed to be used, 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**],[**so_flask**]

Being 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 Django framework supports. It is worth to mention, that two other databases can be used as well - Oracle and MySQL. However, Oracle is ruled out due to its commercial licence, 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 the 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 that is almost fully compatible with the SQL standard and is a de-facto standard in most modern application. It offers a lot of supplementary features, such as complex data types, proper replication and logging, full text search, and many others. In addition to that, PostgreSQL provides extensions which can fit the database to the specific application. [**postgres**] Django has the best support for it, adding a lot of wrappers for Postgre's custom logic.[**django_postgres**]

- **MariaDB**
  MariaDB is a fork of MySQL which adds several advanced features and performance optimizations. One of the prominent aspects of this database are its pluggable storage engine architecture, which allows you to choose different engine per table for the needed workload.[**mariadb**] If set up right, it can also be lightweight and use less system resources than PostgreSQL. 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, which means that the code has to be complied to WebAssembly first (supported by browsers). This adds additional overhead to the development cycle, introduces new toolchains that add another level of complexity and more often than not 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 Frontend of the application is written in JavaScript and only JS frameworks are compared.

Modern JavaScript development is largely dominated by three technologies: React, Vue and Angular.[**frameworkdata**] Each of them introduces a different approach to building frontend applications and comes with its own set of 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 a variety of state management tools and libraries.

  One of the main React's strengths is its ecosystem - it is supported by a wide range of third-party packages, developer tools, and documentation. Additionally, React is maintained by a large team, and a lot of examples of working, 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 in comparison to React or Angular. In some cases it can lead to the need of implementation of features already covered by existing solutions in other frameworks.

- **Angular**
  Angular is a full-stack framework for building web applications - hence, same as 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 which often leads to more robust and structured code. This is a plus for bigger projects, but 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 sheer pace of development that can be achieved with it.

After giving the outline of the client and server side of 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 for quicker access to the content compared to 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 doesn't 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 for 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 it 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 greatly 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 token has all the needed information for user authentication. It can also carry 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 mean of authorization, however, it must be used in congestion with other regular methods.

JWT tokens were chosen as the main 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 amount of API calls needed to get the User information, as some of the information can be embedded inside.

The last important concept that is needed 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 the performance limitations and its inefficiency. P2P techniques(e.g. WebRTC) will not be listed, because the architecture of the application 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 not to be suitable for the logic that will be 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 content-type HTTP responses(technically one never-ending response) with a special 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 for applications where the client initiates most of the interactions, this limitation is not critical. 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 times.

Furthermore, WebSocket traffic is not always reliably supported in all environments. Firewalls and corporate proxies may block or degrade persistent TCP connections, which can affect delivery 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, were message delivery time does not play a big role, it is not suitable for interaction that require 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.

Next chapter Implementation provides the details of the actual implementation of the application. hyperref

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.

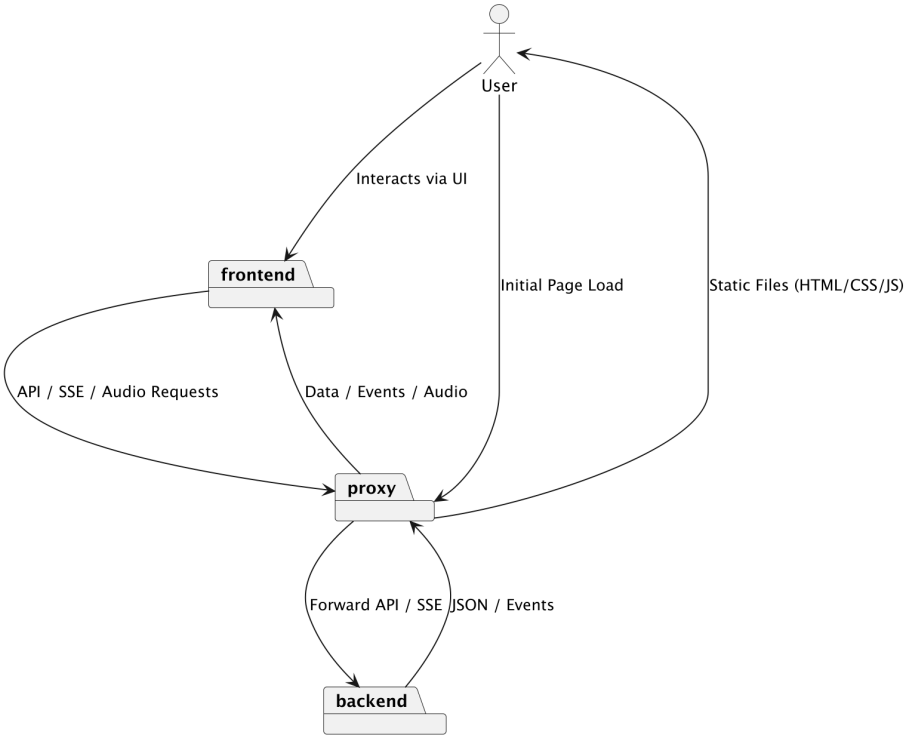


Figure 6.1: System Overview

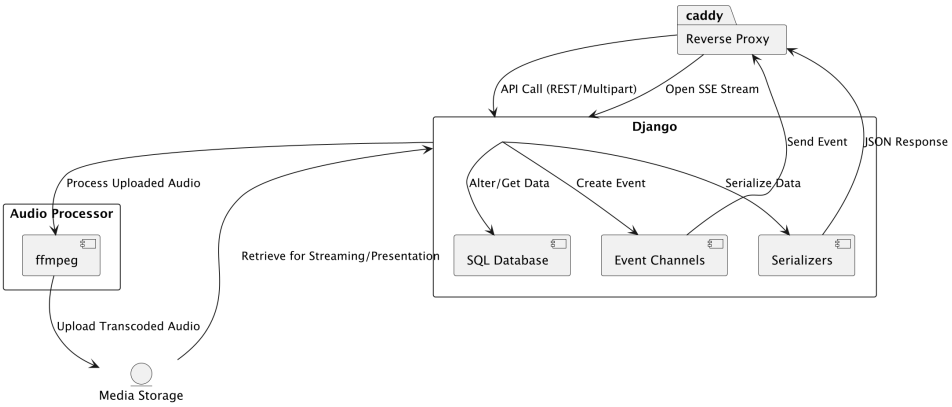


Figure 6.2: Backend

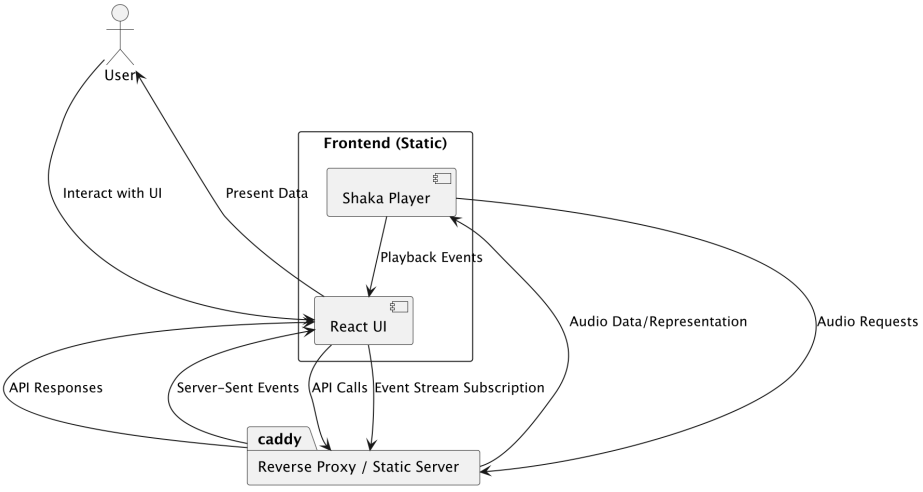


Figure 6.3: Frontend

6.1 Audio Processing, Representation and Transfer

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 or its byte ranges. 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, a 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 MP3, AAC, or Opus, which use perceptual models and compression techniques that aren’t strictly tied to sample

rate or bit depth.

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.

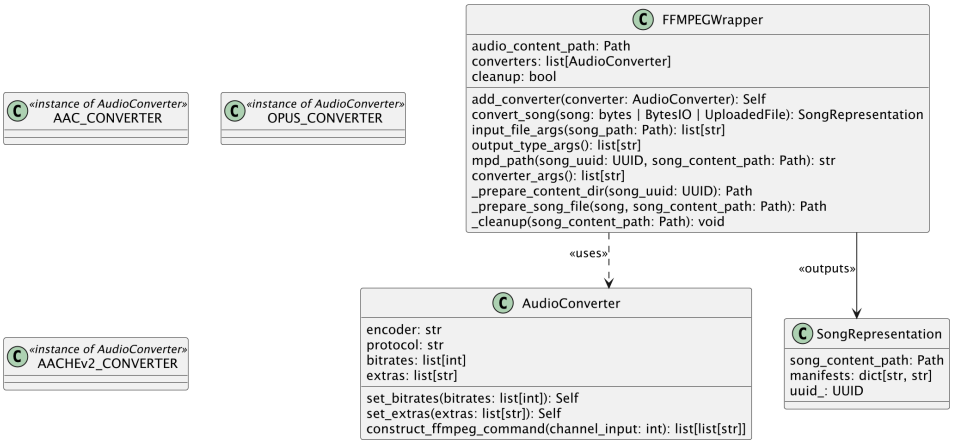


Figure 6.4: FFMPEG Wrapper

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*
```

`-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.

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, so they can later be served to the frontend audio player.

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

6.2 Backend

The description is organized by individual Django *apps*. Each app section follows this structure:

1. **Models** – Define the data schema using Django's ORM.
2. **Serializers** – Convert Django models or native Python types into formats suitable for HTTP transmission (and vice versa).
3. **API Endpoints** – Implemented as *views*, these handle HTTP requests, apply 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.

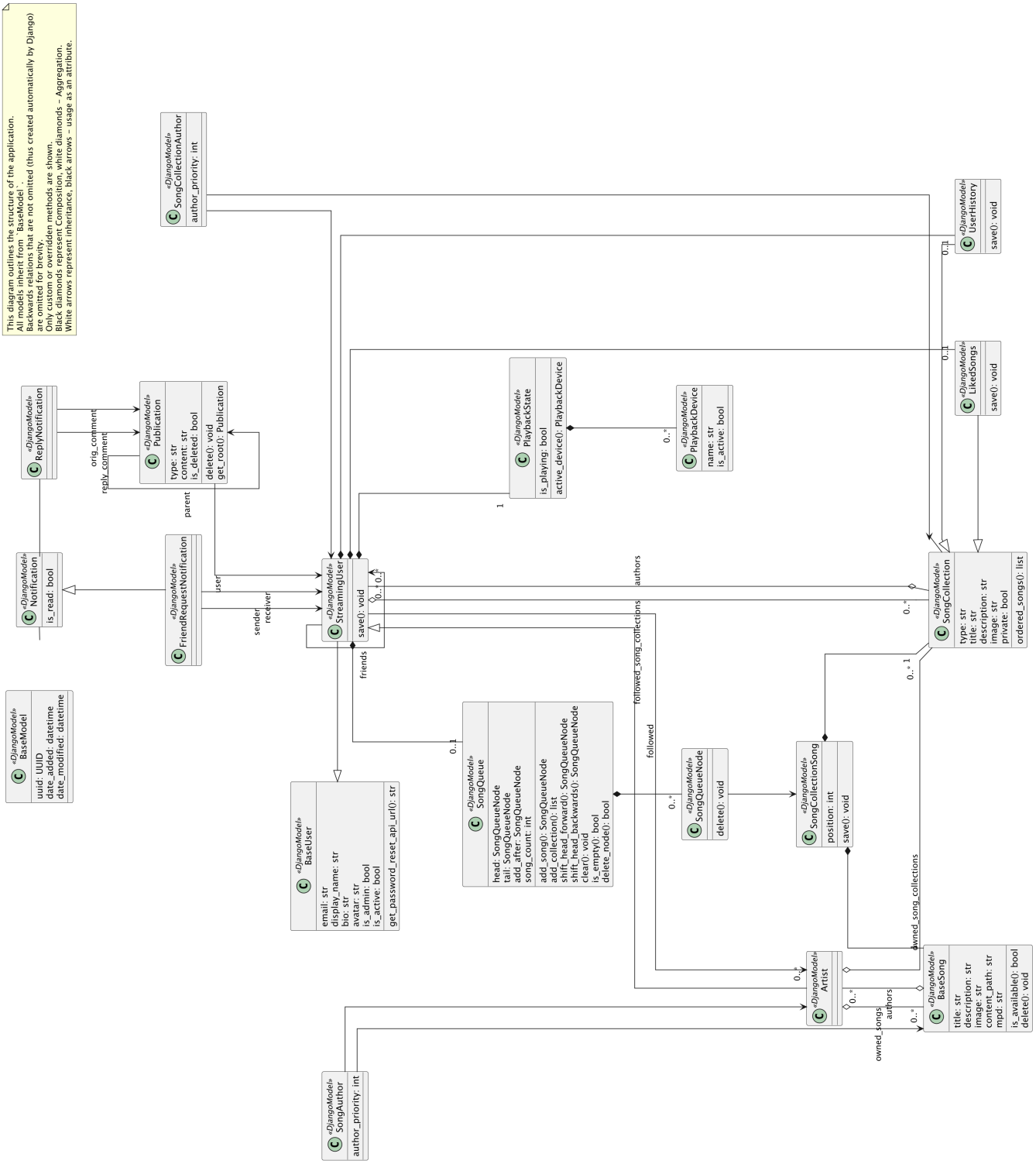


Figure 6.5: Backend Class Diagram

6.2.1 *base*

In this app the abstract `BaseModel` class is defined, all other models inherit from it. It has three attributes - `uuid, date_added, date_modified`. While the latter two are self-explanatory, `uuid` deserves an explanation. 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, since it is much easier to get an existing ID and simply send requests to API endpoints with an incremented ID value.
A serializer for the model is defined here as well, it is also the base class for the serializers of other models.

6.2.2 *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 listened/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. The authentication process of individual HTTP requests is made by adding the token to each request,

then parsing and validating it. After validation the supplied User identifier is extracted and is used to find the existing User record to which the request belongs. Thus, the `uuid` of the User is added to the token payload, and the authorization logic, and it is customized to use UUID instead of ID for the user lookup. The usage of UUID in the token payload will be described in more detail in the frontend section.

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, could be deduced from their name.
UserEventViewSet is a special view used to implement the SSE connection, the details are provided in the next subsection.

6.2.3 *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 mostly used throughout the system 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 used for models that are represented 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, specifically 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 it safe to move on to the main part of the application - the *streaming* package.

6.2.4 *streaming*

say about playback

Chapter 7

Summary and Conclusion

Sources