



Assignment of bachelor's thesis

Title:	Development of MusicMates Web Application
Student:	Kirill Alekhnovich
Supervisor:	Ing. Marek Suchánek, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

Spotify is a widely used and popular music and streaming service; however, it does not provide direct user/library metrics and comparisons useful for friends. On the other hand it provides an API so it is possible to build such applications externally. The goal of this thesis is to design and implement a web application that will take advantage of Spotify API and provide various metrics tracking for users, sharing them, and comparing them among users.

- Analyse Spotify features and API, identify useful resources for retrieving information for metrics and statistics.
- Research existing solutions that use Spotify API for the same or similar goals.
- Set requirements for your own solution and prepare use cases.
- Design the solution with respect to best practices of software engineering. Focus on sustainability and extensibility of the application.
- Implement as a client-server web application with Spring Boot / Kotlin for backend and Vue.js / TypeScript for frontend.
- Test the application based on the design.
- Evaluate the solution and outline potential future improvements.

Bachelor's thesis

DEVELOPMENT OF MUSICMATES WEB APPLICATION

Kirill Alekhnovich

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Marek Suchánek, Ph.D.
February 14, 2024

Czech Technical University in Prague
Faculty of Information Technology

© 2024 Kirill Alekhovich. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Alekhovich Kirill. *Development of MusicMates Web Application*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

Acknowledgments	vi
Declaration	vii
Abstract	viii
List of abbreviations	ix
Introduction	1
Goals	2
1 Analysis	3
1.1 Spotify Web API	3
1.1.1 Endpoints	3
1.2 Analysis of existing solutions	5
1.2.1 Last.fm	5
1.2.2 stats.fm	7
1.2.3 Stats for Spotify	9
1.2.4 Obscurify	9
1.2.5 musictaste.space	10
1.2.6 Summary of existing solutions	11
1.3 Requirements analysis	12
1.3.1 Functional requirements	13
1.3.2 Non-functional requirements	13
2 Design	15
2.1 Technologies	15
2.1.1 Frontend	15
2.1.2 Backend	16
2.1.3 Database	17
2.2 Architecture	18
2.2.1 Presentation tier	18
2.2.2 Application tier	19
2.2.3 Data tier	19
2.3 API	19
2.3.1 Endpoints	21
2.4 Graphical User Interface	23
2.4.1 Main page	23
2.4.2 User profile	24
2.4.3 Album page	26
2.4.4 Artist page	26
2.4.5 Track page	27

3	Implementation	28
3.1	Backend implementation	28
3.1.1	Computing statistics	28
3.1.2	Updating playbacks	28
3.1.3	Mapping platform objects	29
3.1.4	Calculating compatibility	30
3.1.5	Security	30
3.1.6	Lyrics fetching	30
3.2	Frontend implementation	31
3.2.1	Composables	31
3.2.2	TanStack Query	31
4	Testing	33
4.1	Types of tests	33
4.1.1	Unit testing	33
4.1.2	Manual testing	33
4.2	Results of tests	34
5	Evaluation	35
5.1	Usability	35
5.2	Comparison with existing solutions	35
5.3	Future improvements	35
5.3.1	Extended statistics	35
5.3.2	Profound preferences comparison	36
5.3.3	Support more music services	36
5.3.4	Responsive design	36
5.3.5	Custom profile images	36
5.3.6	Complete album fetching	36
5.3.7	Mobile applications	36
6	Conclusion	37
	Contents of the attachment	42

List of Figures

1.1	Last.fm user page [17]	6
1.2	Last.fm report's most listened artists, albums and tracks [17]	6
1.3	Last.fm report's charts and diagrams [17]	7
1.4	stats.fm user profile [22]	8
1.5	stats.fm friends comparison [22]	8
1.6	Obscurify rating [27]	9
1.7	Obscurify tracks' analysis [27]	10
1.8	music taste.space's compatibility report [29]	11
1.9	MoSCoW prioritization [31]	12
2.1	Three tier architecture [52]	18
2.2	Database model	20
2.3	Main page	23
2.4	Overview tab	24
2.5	Stats tab	25
2.6	Friends tab	25
2.7	Album page	26
2.8	Artist page	26
2.9	Track page	27

List of Tables

1.1	Implemented functionality in existing applications	12
1.2	Functional requirements	13
1.3	Non-functional requirements	14
2.1	REST principles [53]	21

List of code listings

1	Updating playbacks	29
2	Mapping objects	30
3	Lyrics fetching	30
4	Date to string composable	31
5	Duration to string composable	31
6	Tanstack Query wrapping	32
7	Get playbacks function	32

I would like to thank my supervisor Ing. Marek Suchánek, Ph.D. for his guidance and insightful suggestions on writing this thesis. A big thanks also belongs to my family and friends for their consistent support throughout my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on February 14, 2024

Abstract

The subject of this bachelor's thesis is the design and implementation of a web application, that provides its users with statistics of their listening activity on different music streaming services. It uses Spotify Web API for user data fetching and conducts an analysis based on the user's playback history. The resulting software consists of a Kotlin Spring backend application and a web client implemented using TypeScript and the Vue.js framework. It allows its users to review their preferences in music by providing corresponding statistics, and to compare musical preferences with their friends.

Keywords web application, music, statistics, Spotify, Kotlin, Spring, Vue, TypeScript

Abstrakt

Předmětem této bakalářské práce je návrh a implementace webové aplikace která umožní svým uživatelům sledovat statistiky jejich aktivity na různých hudebních streamovacích službách. Tato aplikace používá Spotify Web API pro načítání uživatelských dat a provádí analýzu založenou na historii přehrávání uživatele. Výsledný software se skládá z backendové Kotlin Spring aplikace a webového klienta implementovaného pomocí TypeScript a frameworku Vue.js. Aplikace umožňuje svým uživatelům sledovat své preference poskytnutím odpovídajících statistik, a porovnávat hudební preference se svými kamarády.

Klíčová slova webová aplikace, hudba, statistika, Spotify, Kotlin, Spring, Vue, TypeScript

List of abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
CSS	Cascading Style Sheets
DI	Dependency Injection
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IOC	Inversion of Control
ISRC	International Standard Recording Code
JS	JavaScript
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
JWT	JSON Web Token
REST	Representational State Transfer
TS	TypeScript
UI	User Interface
URL	Uniform Resource Locator
UX	User Experience

Introduction

We live in a time when music streaming services have become an essential part of people's lives. Every day millions of people listen to their favorite songs on various music platforms, constantly find new ones, and create playlists based on their musical preferences. Moreover, these streaming services provide artists with an opportunity to share songs they create with large numbers of people and significantly expand their audience.

Despite all this, modern streaming services do not provide any social interactions across their user base. They don't give their users a way to view listening statistics, share their listening history, or compare it with friends. Lack of such functionality makes it more difficult for users to analyze their musical preferences as well as to find new songs and artists they may like.

Based on this, the result of this thesis will be useful for active users of music services who want to better understand what songs they and their friends like. Thanks to the final application, users will be able to enrich their library with new songs, easily share their musical preferences with other people, and find new friends among music lovers.

I chose this topic because this issue has not yet been resolved sufficiently enough, and also because implementation of such a service will help me apply the skills acquired during my studies at the university in practice.

The objective of this thesis is to conduct an analysis of existing solutions, define functional and non-functional requirements for the application, and implement software based on the set requirements. The resulting application will allow users to interact with one another, see playback history and most-listened artists of other people, and compare musical preferences with them.

This thesis is divided into six chapters. The first chapter focuses on analyzing existing solutions and identifying functional and non-functional requirements for the application. The second chapter discusses the design and architecture of the future application. The third chapter contains a description of the development of all parts of the application, describes problems that arose during implementation, as well as their solutions. The fourth chapter focuses on how different parts of the application were tested. The fifth chapter describes the usability of the resulting application, compares it with existing solutions, and presents ideas for possible improvements of the application in the future. The last chapter summarizes the work done and verifies that all stated goals are fulfilled.

Goals


The main goal of this thesis is to develop a web application that will allow its users to better understand their musical preferences by showing statistics on their listening activity, as well as provide users with an opportunity to follow their friends and compare musical preferences with each other.

To achieve this goal, a set of subgoals has to be done. It includes analysis, design, implementation, testing, and evaluation of the application.

During the analysis stage, research on the Spotify API will be conducted. In addition, similar existing solutions will also be examined. This analysis will be summed up by corresponding functional and non-functional requirements set based on the MoSCoW principle.

Other subgoals are to design the application with a focus on its sustainability and extensibility, and then implement and test the application based on the set requirements and the design.

In conclusion, there will be an evaluation of the solution and a list of potential future improvements.



Chapter 1

Analysis

This chapter is dedicated to identifying useful resources for retrieving data for metrics and statistics from the Spotify Web API, analyzing other existing solutions that provide functionality similar to the goals stated in the previous section, and defining functional and non-functional requirements for the application.

1.1 Spotify Web API

Spotify Web API enables the creation of applications that can interact with Spotify's streaming service, such as retrieving content metadata, getting recommendations, creating and managing playlists, or controlling playback. [1]

1.1.1 Endpoints

To understand how to leverage the full potential of the Spotify Web API, it is necessary to examine the endpoints it provides.

Spotify Web API is a RESTful API with different endpoints that return JSON metadata about music artists, albums, and tracks, directly from the Spotify Data Catalogue. Web API returns JSON in the response body, however, some endpoints return just the HTTP status code instead. [2]

During the research stage, endpoints were examined and the following ones were marked as potentially useful in the future application:

1.1.1.1 Get Album

Get Album endpoint returns an album with its metadata. The request must include the Spotify ID of the album. The response to the request is an album object. It contains fields such as type, images, name, release date, artists, tracks, etc. [3]

1.1.1.2 Get Several Albums

Get Several Albums endpoint returns a list of albums with their metadata. The request must include comma-separated Spotify IDs of the albums in order to return a correct list of albums. [4]

1.1.1.3 Get Artist

Get Artist endpoint returns an artist with its metadata. The request must include the Spotify ID of the artist. The response to the request is an artist object. It contains fields like genres, images, name, etc. [5]

1.1.1.4 Get Several Artists

Get Several Artists endpoint returns a list of artists with their metadata. The request must include comma-separated Spotify IDs of the artists in order to return a correct list of artists. [6]

1.1.1.5 Get Track

Get Track endpoint returns a track with its metadata. The request must include the Spotify ID of the track. The response to the request is a track object. It contains fields like album, artists, images, name, duration, etc. [7]

More importantly, track object has a field called ISRC. The International Standard Recording Code (ISRC) is a code used to uniquely identify sound recordings and music video recordings, therefore different recordings, edits, and remixes of the same work each have their own ISRC. [8]

It is worth noting that Spotify is not the only streaming service that provides this code for the tracks. For example, Apple Music also exposes tracks' ISRCs, and allows to search tracks by them as well [9, 10]. Therefore, tracks between different music streaming services can be mapped, and the application can support a variety of music platforms.

1.1.1.6 Get Several Tracks

Get Several Tracks endpoint returns a list of tracks with their metadata. The request must include comma-separated Spotify IDs of the tracks in order to return a correct list of tracks. [11]

1.1.1.7 Search for Item

This endpoint gets information from the Spotify catalog that matches the search query. It has two required attributes: a query and a type.

The query attribute has a set of available filters: album, artist, track, year, upc, tag, isrc, and genre. Each field filter only applies to certain result types.

The type is another required parameter. It is a comma-separated list of item types to search across. The admissible values are album, artist, playlist, track, show, episode, and audiobook.

The response of this API call is a list of objects of the corresponding types specified in the type field of the initial request. [12]

1.1.1.8 Get User's Top Items

This endpoint returns current user's top artists or tracks based on calculated affinity. It can provide top items for three different time ranges: short-term (last 4 weeks), medium-term (last 6 months), and long-term (last several years).

This endpoint requires a user to be authorized with *user-top-read* scope. [13]

1.1.1.9 Get Recently Played Tracks

As the name suggests, this endpoint is used for fetching user's playback history. It returns a list of tracks alongside with their timestamps and playing context. It is important to note that this endpoint can only return last 50 playbacks.

This endpoint requires a user to be authorized with *user-read-recently-played* scope. [14]

1.1.1.10 Get Currently Playing Track

This endpoint returns an object currently being played on user's Spotify account. It can be either a track or an episode. The endpoint also provides data about the context and device on which this object is being played.

This endpoint requires a user to be authorized with *user-read-currently-playing* scope. [15]

1.1.1.11 Get Track's Audio Features

Track's Audio Features endpoint is used for track analysis. It expects the Spotify ID of the track and returns different metrics about the track. These metrics include loudness, danceability, instrumentalness, and other rates. It also provides modality of the track and its tempo in Beats Per Minute (BPM). [16]

It is worth noting that some of the endpoints above have not been used in the application. However, they are used in several existing solutions and therefore are mentioned here.

1.2 Analysis of existing solutions

To set functional and non-functional requirements correctly, research on other similar solutions must be conducted. This section contains an overview of existing applications that process data from music streaming services and provide statistics. Based on responses from search engines for queries related to this topic, the following applications have been chosen for the analysis: Last.fm, stats.fm, Stats for Spotify, Obscurify, and musictaste.space.

1.2.1 Last.fm

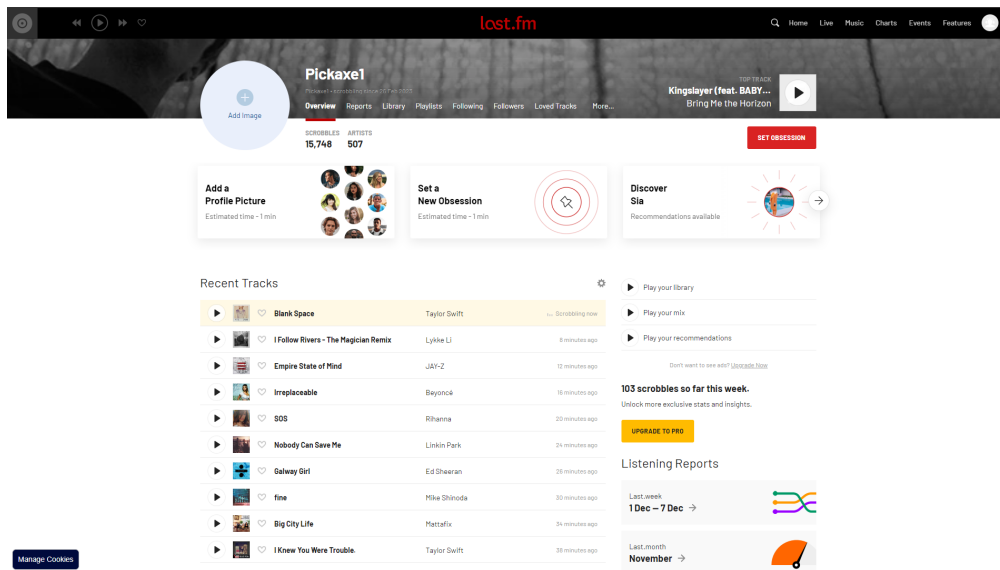
One of the most well-known applications that is capable of analyzing user's musical preferences is Last.fm [17]. Its pages state that it has been around for nearly 20 years, it is used by millions of users, and its database contains more than a billion unique tracks [18, 19]. Last.fm has a website, iOS and Android applications in App Store and Google Play respectively.

The way this application works is by examining user's listening history. It introduces so-called scrobbles and scrobbling. Scrobble is a record in the Last.fm database that stores what track user has listened to and when, and scrobbling is the process of sending this information from music players, devices, and services to Last.fm. Apps and plugins that do scrobbling are called scrobbles. Based on scrobbles, Last.fm generates weekly, monthly, and yearly statistics for its users and recommends new music. [19]

It is worth noting that Last.fm supports scrobbling from a large set of music players. However, due to different APIs of these players, Last.fm cannot have one versatile solution for accessing user's listening history on all platforms. For example, to start scrobbling from Spotify it's enough just to link a Spotify account to Last.fm, but for other popular services like Apple Music, YouTube Music, SoundCloud, and others, it is mandatory to use one of the available scrobbles. [20, 21]

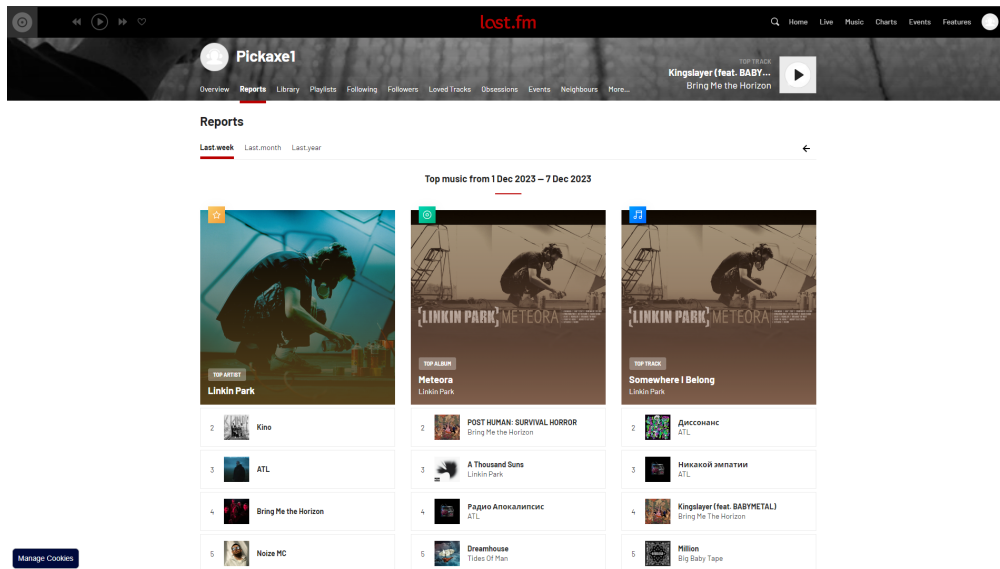
Speaking of the functionality, Last.fm allows to follow other users, see their reports, and what they are listening to. It also shows a compatibility between friends, which, however, does not

contain a detailed comparison of musical preferences, but only a few common artists (if there are any) and a level of compatibility.

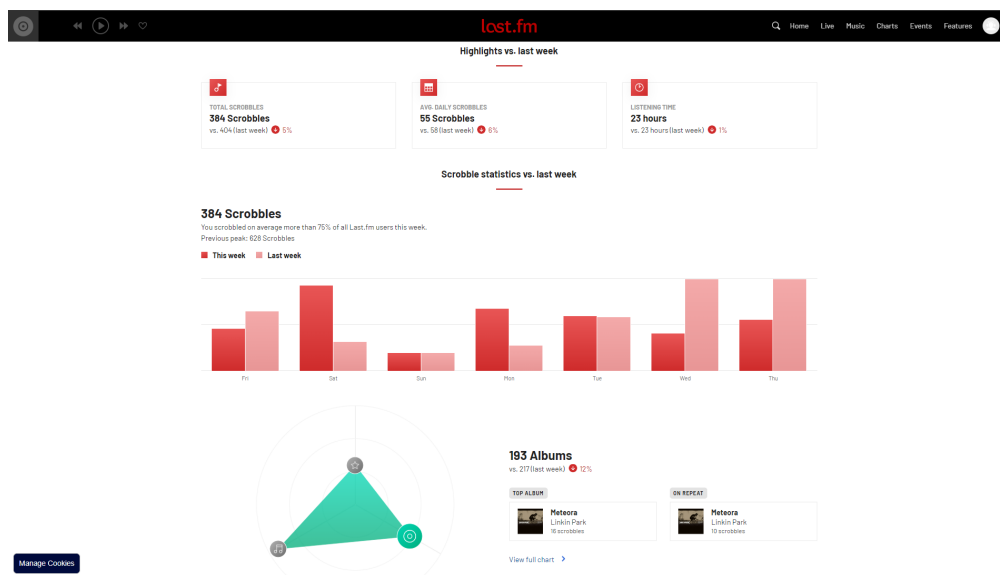


■ Figure 1.1 Last.fm user page [17]

Each week Last.fm updates reports for all users. Reports consist of lists of the most-listened artists, albums, and tracks for a given period, as well as different charts and diagrams representing trends in user activity.



■ Figure 1.2 Last.fm report's most listened artists, albums and tracks [17]



■ **Figure 1.3** Last.fm report's charts and diagrams [17]

Any engineering solution comes with its own advantages and disadvantages, and Last.fm is not an exception.

Pros:

- Supports import of listening history from different streaming services and applications.
- Stores a record for each playback, therefore can generate more precise statistics.
- Provides detailed reports for different time periods.
- Allows to check friends' listening history and reports.
- Recommends new music based on what the user listens to.

Cons:

- Has plenty of extra features that cause the application to look more complex.
- Starts analyzing user's preferences only after registration (so it becomes impossible to compare them with someone who does not have an account on Last.fm).

1.2.2 stats.fm

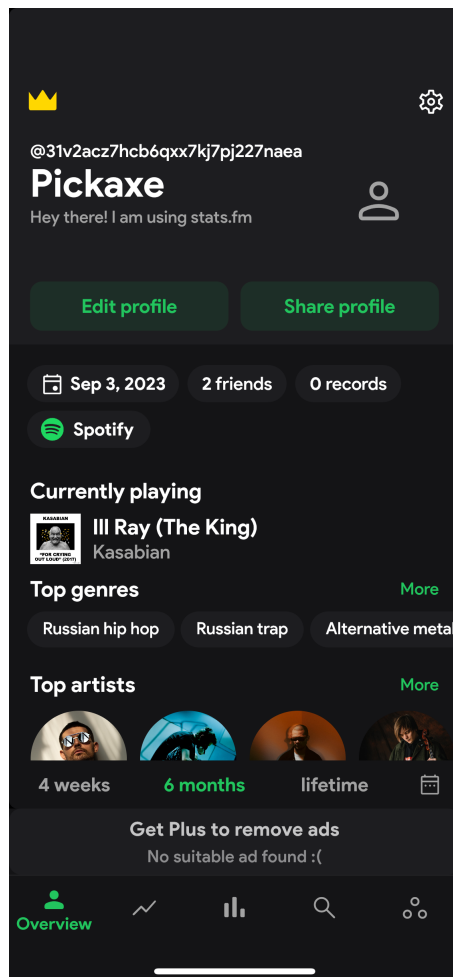
One more platform providing similar functionality is stats.fm. Despite being targeted at Spotify users only, it still has millions of users and hundreds of millions of tracks in its database. Like Last.fm it also has a web version, as well as iOS and Android applications. [22]

This application has two types of users: free and plus [23]. To acquire a plus status, a one-time purchase must be made. It is important because the application behaves differently for these users.

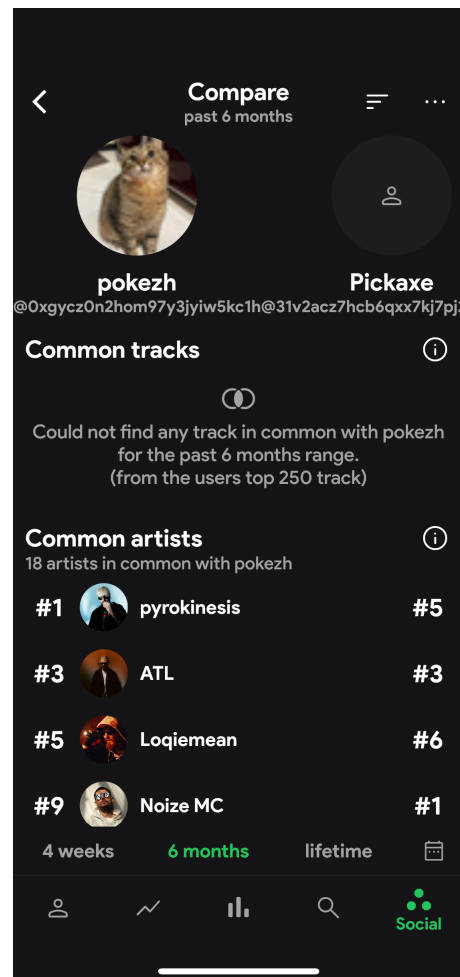
For free users, top artists and tracks are fetched directly from the Spotify API's top items endpoint described in Section 1.1.1.8. Having that said, users can only view their top 50 tracks and artists for 3 predefined time ranges (4 weeks, 6 months, several years). In addition, only 50 recently played tracks can be viewed due to the limitations of Spotify API described in Section 1.1.1.9. On the other hand, plus users can enable the history synchronization option, which automatically keeps user's streaming history up to date every 60 minutes. Moreover, plus

users can request a lifetime listening history from Spotify and export it to stats.fm in a JSON format. This way, stats.fm can provide more diverse and accurate statistics as it calculates them based on the playback count instead of relying on Spotify algorithms. [24, 25]

In addition to this functionality, stats.fm also has an option of adding friends. Unlike Last.fm, it allows to compare top artists, tracks, and genres between users, but it does not show the compatibility score. Friends can also send messages to each other.



■ Figure 1.4 stats.fm user profile [22]



■ Figure 1.5 stats.fm friends comparison [22]

Pros:

- Allows importing a lifetime playback history (has to be requested from Spotify).
- Offers a detailed comparison of musical preferences.
- Can compare musical preferences right after the registration.

Cons:

- Targeted only at Spotify users.
- Free users can only view 50 recently played songs, and top 50 tracks and songs (therefore no metrics are provided).

1.2.3 Stats for Spotify

One of the first websites that pop up in search engines for the query "Spotify stats" is Stats for Spotify [26]. Despite being so high in the search, it does not provide any statistics. It shows recently played tracks, top artists, tracks, and genres just by querying the Spotify endpoints mentioned in Section 1.1.1.8 and Section 1.1.1.9, and then represents results within its UI.

Pros:

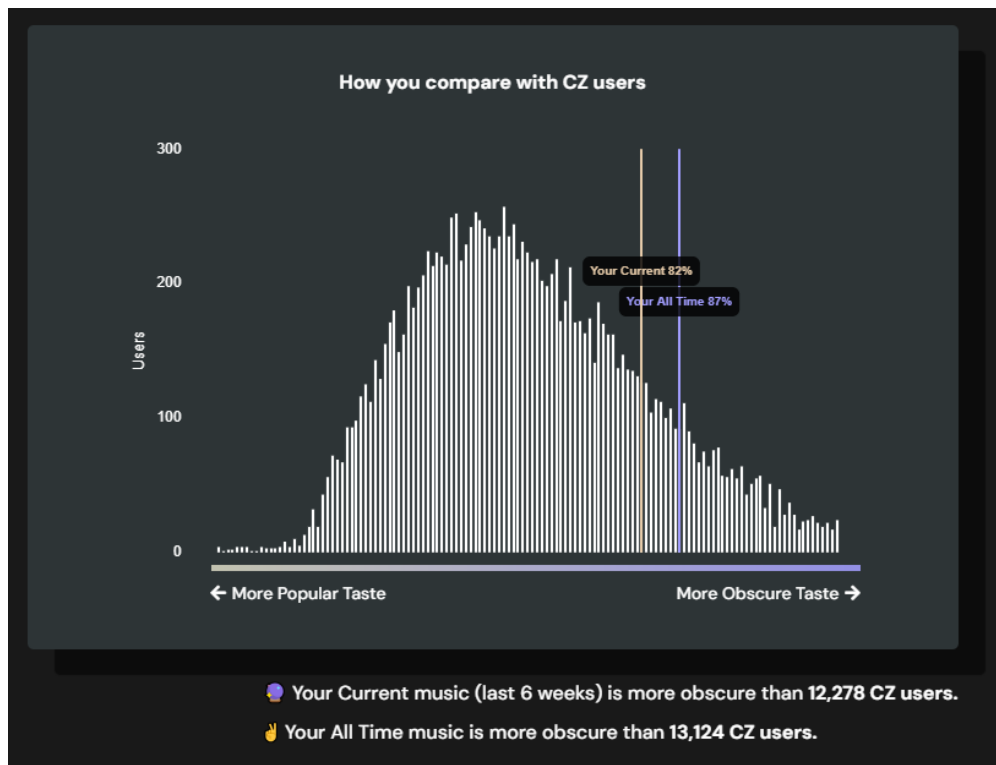
- Displays instant statistics.
- Good for those who want to see their stats just once.

Cons:

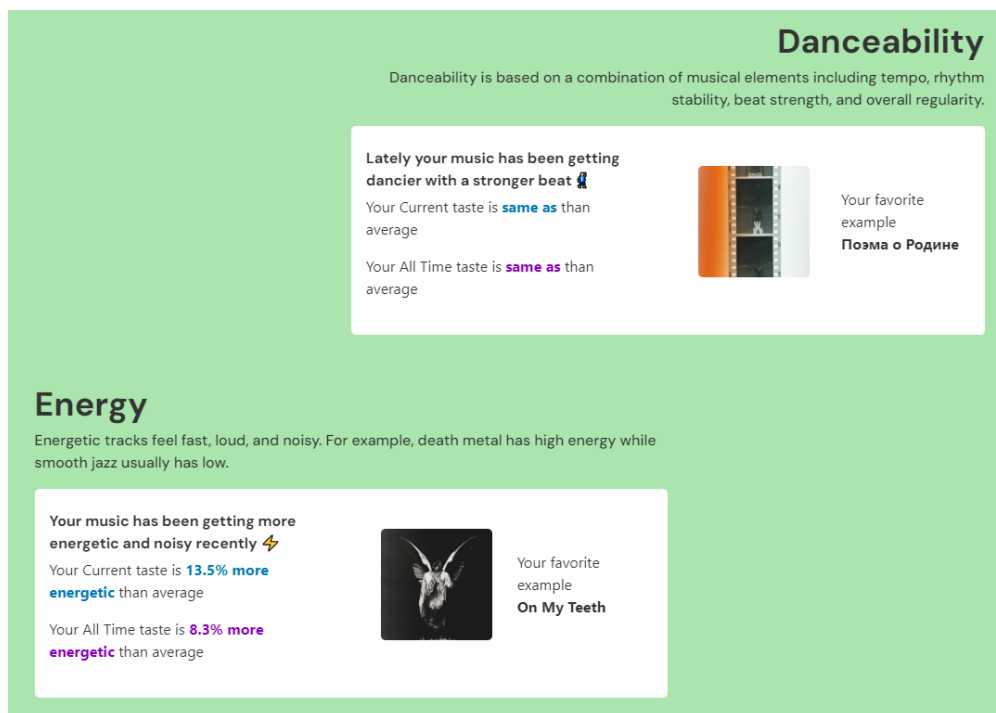
- Targeted only at Spotify users.
- Does not provide any profound statistics.
- Does not provide a way to compare stats with friends.

1.2.4 Obscurify

Obscurify [27] is the application that uses Spotify Web API to display listening history, determine top genres, and recommend songs users might like [28]. It uses endpoints mentioned in Section 1.1.1.8 and Section 1.1.1.11 to fetch data and display happiness, energy, danceability, and acousticness of the songs user listens to. Obscurify also shows an obscurity rating, which measures the uniqueness of user's musical preferences. It saves snapshots of top tracks and artists each time user requests them. This way user can review his older stats.



■ Figure 1.6 Obscurify rating [27]



■ **Figure 1.7** Obscurify tracks' analysis [27]

Pros:

- Displays instant statistics.
- Provides additional features such as obscurity rating, happiness, energy, danceability, and acousticalness.
- Allows to share personal stats.

Cons:

- Targeted only at Spotify users.
- Does not provide a way to compare stats between users.

1.2.5 [musictaste.space](#)

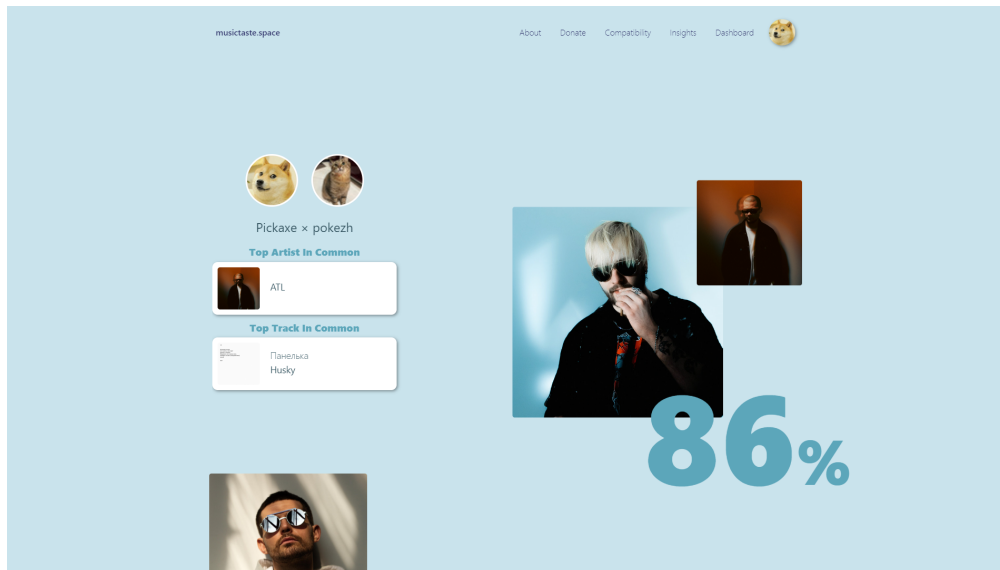
One more option for musical preferences analysis is [musictaste.space](#) [29]. This application helps users gain insights into their Spotify listening habits and compare their musical preferences with their friends. It shows top artists, tracks, and genres as a part of the statistics and also generates a link that allows other people to check their compatibility with the user. Compatibility calculations are mainly based on common genres and artists, but also rely on common tracks as well. Matches in less popular genres give more points to the total compatibility score. [30]

After logging in with Spotify, [musictaste.space](#) offers to view insights, compare musical preferences with others, and make a playlist.

Insights contain the same data as Obscurify (see Section 1.2.4), including the obscurity rating (it is mentioned that this rating is actually calculated using Obscurify's algorithm).

The compatibility tab contains a link for sharing calculated insights and comparing them with other users. This link can include a Spotify username and image or be completely anonymous. Another user can open this link to generate a compatibility report. It contains a percentage

score of the compatibility and a complete breakdown on what makes that user compatible. This breakdown includes top common genres, as well as top common artists and tracks for different time periods.



■ **Figure 1.8** music taste.space’s compatibility report [29]

Pros:

- Displays instant statistics.
- Offers detailed comparison of musical preferences.
- Can compare musical preferences anonymously.

Cons:

- Targeted only at Spotify users.

1.2.6 Summary of existing solutions

For comparing the solutions mentioned above Table 1.1 has been made. According to it, only Last.fm can fetch user’s listening history from different music streaming services, however, it can not provide statistics right away as user has to be registered for a while in order to gain more playbacks. Other popular applications have this option, but they rely on the data from the Spotify API, and do not provide support for other music platforms. These data are also limited by predefined time ranges (4 weeks, 6 months, several years), which deprives the opportunity to show statistics for manually chosen time ranges. Several applications can calculate compatibility between users, however, while Last.fm shows just the rate, stats.fm and music taste.space provide a breakdown that specifies compatibility more precisely.

Application \ Functionality	Support of multiple platforms	Instant statistics	Own way of stats computing	Compatibility calculation	Lifetime playbacks import
Last.fm	+	-	+	+	-
stats.fm	-	+	+	+	+
Stats for Spotify	-	+	-	-	-
Obscurify	-	+	-	-	-
musictaste.space	-	+	-	+	-

+ – option available
 – – option is missing

■ **Table 1.1** Implemented functionality in existing applications

1.3 Requirements analysis

Based on the thesis assignment and the analysis in Section 1.2, requirements for the application will be set. These requirements will be divided into functional and non-functional ones, and also split into four categories according to the MoSCoW prioritization method [31]: must have, should have, could have, and will not have (see Figure 1.9). Each requirement number will be annotated either with *F* if it's a functional requirement, or with *NF* if it's a non-functional requirement.

MoSCoW prioritization



■ **Figure 1.9** MoSCoW prioritization [31]

1.3.1 Functional requirements

Functional requirements define the features that particular software must provide to the end users of the application [32]. Requirements *F1*, *F2*, and *F3* are based on the thesis assignment, and therefore are in the "must have" category. Table 1.2 sums up the content of this section.

[F1] Statistics compute – M

The application must compute statistics based on user's listening history and represent them within its UI.

[F2] Compatibility calculation – M

The application must be able to calculate the compatibility between each two users using computed statistics.

[F3] Adding friends – M

The functionality of adding and removing friends must be implemented. Adding a friend will help to navigate to his page more quickly.

[F4] Own stats generating mechanism – S

Own stats generation mechanism will help to compute unified statistics for multiple music streaming services in the future. It is better than relying on external statistics fetched from corresponding music platforms as each platform computes statistics differently and some of the streaming services may not expose user's top items data.

[F5] Support of different music services – C

The application should be designed in a way that would allow support of multiple music streaming services. It is not mandatory to support them in the current state, but it should be easy to add new platforms in the future.

[F6] Lifetime listening history import – W

Despite importing a lifetime listening history from the music service being a useful feature, the majority of users most likely will not use it due to its complexity and necessity to request listening history from music platforms. This option may be considered in the future, but it is surely not a priority.

No.	Functional requirement	Priority
F1	Statistics compute	M
F2	Compatibility calculation	M
F3	Adding friends	M
F4	Own stats generating mechanism	S
F5	Support of different music services	C
F6	Lifetime listening history import	W

■ **Table 1.2** Functional requirements

1.3.2 Non-functional requirements

Non-functional requirements define a set of constraints on the design of the system. They also help to verify the performance of the software [32]. Requirements *NF1*, *NF2*, *NF3*, and *NF4* are based on the assignment of the thesis, and therefore are in the "must have" category. Table 1.3 sums up the content of this section.

[NF1] Implementation of the backend using Spring Boot and Kotlin – M

The server part must be implemented in Kotlin programming language using Spring Boot framework.

[NF2] Implementation of the frontend using Vue.js and TypeScript – M

The client part must be a web application implemented in TypeScript using Vue.js library.

[NF3] Extensibility – M

The resulting application must be extensible in order to make future updates and development easier.

[NF4] Testing – M

The application must be properly tested. The server part must be covered with unit tests and the client must be manually tested.

[NF5] User interface and experience – S

The design of the application should be user-friendly. The user interface should be designed in a way that makes user interactions with the application as simple and as intuitive as possible.

[NF6] Multiple themes – C

The application may support light and dark themes in the future. It makes the use of the application more pleasant and enjoyable for users.

[NF7] Localization – C

Localization is another nice feature to have in the application that can broaden its audience. It is not required in the current iteration of the project, but could be added in the future.

[NF8] Mobile application – W

For now, the development of the mobile application is not a priority as it requires significant resources and does not help to identify whether the application works as intended.

No.	Non-functional requirement	Priority
NF1	Implementation of the backend using Spring Boot and Kotlin	M
NF2	Implementation of the frontend using Vue.js and TypeScript	M
NF3	Extensibility	M
NF4	Testing	M
NF5	User interface and experience	S
NF6	Multiple themes	C
NF7	Localization	C
NF8	Mobile application	W

■ **Table 1.3** Non-functional requirements



Chapter 2

Design

This chapter is dedicated to the design of the application. It describes the architecture and technologies used on the client part, the server part, and the database. This chapter also contains a description of the implemented API and its most-used endpoints. This description is followed by a presentation of the graphical user interface (GUI) of the application.

2.1 Technologies

The whole application is split into 3 parts – the frontend part, the backend part, and the database. This section describes the technologies used in each of these 3 parts.

2.1.1 Frontend

Frontend refers to the User Interface (UI) and the User Experience (UX) of a web application. It involves visual elements of the application, including layout, design, and their interactivity.

Most commonly used technologies to build the frontend part of the application are HTML, CSS, and JavaScript. HTML is used to define the content of pages of the web application, CSS adds styling to the elements on these pages, and JavaScript is used to define the logic of interactions with these elements. Nowadays, different frontend frameworks are also widespread in the process of building web applications. They improve the overall performance of a website and offer a set of features that make frontend development easier. [33]

This section describes technologies used on the frontend side of the application.

2.1.1.1 TypeScript

TypeScript is an open-source strongly typed programming language developed by Microsoft. It extends JavaScript by bringing types, which makes it easier for the developer to catch type-related errors on the spot. By adding additional syntax to JS, TypeScript offers tighter integration with code editors. TypeScript code can also be converted into JavaScript so that it can be run anywhere JavaScript runs. [34]

2.1.1.2 Vue.js

Vue is an approachable, performant, and versatile framework for building user interfaces. It is implemented in TypeScript and provides first-class TypeScript support. Vue also builds on

top of HTML, CSS, and JS/TS and provides a declarative and component-based programming model. [35, 36]

Despite the fact that individual preference plays a crucial role in a comparison of different frontend frameworks, Vue has a set of acknowledged advantages over other popular frameworks like React or Angular, which include its simplicity, flexibility, and a shallower learning curve. [37]

2.1.2 Backend

Backend is responsible for processing and managing data, performing calculations, and facilitating communication between different components of the system.

This section describes technologies used on the backend side of the application.

2.1.2.1 Kotlin

Kotlin is a cross-platform, statically typed, high-level programming language developed by JetBrains. It is designed to fully interoperate with Java, and mainly targets the JVM, but also compiles to JavaScript and native code. [38, 39]

Besides, Kotlin has a set of advantages over Java, which include:

- More concise syntax.
- Null safety and type inference.
- Better support for functional programming.

Having that said, Kotlin can be considered as a legitimate option for the server side of the application.

2.1.2.2 Spring

Spring is the most popular Java framework. It can be used in a variety of ways: in microservice architectures, in cloud, reactive, and serverless solutions. It is based on the Inversion Of Control (IOC) and Dependency Injection (DI) principles. Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications. It is designed in a way that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments. [40, 41]

In 2017, Spring team introduced Kotlin support starting with version 5.0 [42, 43]. Having that said, it is now possible to create Spring applications with Kotlin, and this opportunity is leveraged in the application.

2.1.2.3 Spring Boot

Spring Boot simplifies the process of creating production-grade Spring based applications. It allows to create stand-alone applications by automatically configuring Spring and other 3rd party libraries. It also provides production-ready features such as metrics, health checks, and externalized configuration. [44]

2.1.2.4 Spring Security

Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications. [45]

Spring Security is a framework that focuses on providing both authentication and authorization to Java applications. One of the main advantages of Spring Security is that it helps to easily extend the application to meet custom security requirements. [45]

It also provides a set of different features. The most vital ones are:

- Comprehensive and extensible support for both authentication and authorization.
- Protection against attacks like session fixation, clickjacking, cross-site request forgery, etc.
- Optional integration with Spring Web MVC.

2.1.2.5 Gradle

Gradle is an open-source build tool that makes it easier to build, automate, and deploy applications. It is trusted by millions of developers around the world and has first-class support in all major IDEs. [46]

Compared to Maven (which is another commonly used build tool), Gradle offers better performance by implementing mechanisms of work avoidance and incrementality. [47]

Top 3 features that make Gradle a more suitable option than Maven are:

- **Incrementality.** Gradle avoids redundant work by tracking input and output of tasks and only running what is necessary.
- **Build Cache.** Gradle reuses the build outputs of any other Gradle build with the same inputs.
- **Gradle Daemon.** Gradle runs a long-lived background process that keeps build information in memory.

Moreover, Gradle provides a Kotlin DSL, that brings the elegance and type-safety of Kotlin to automation. [48]

2.1.3 Database

A database is a collection of organized stored data. Its purpose is to ease the process of managing, retrieving, and updating stored data.

There are 2 main types of databases – relational (or SQL) databases, and non-relational (or NoSQL) databases. The key difference between these types is the way how the data is stored. Relational databases store data in tables, while non-relational databases use other structures such as documents, key-value pairs, or any other proper way of storing data. [49]

Due to the reliability achieved by compliance with the ACID properties, as well as data accuracy and simplicity of relational databases, an SQL database has been chosen for the application.

2.1.3.1 PostgreSQL

PostgreSQL is a powerful, open-source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads. PostgreSQL has earned a strong reputation for its proven architecture, reliability, data integrity, robust feature set, and extensibility. PostgreSQL runs on all major operating systems and has been ACID-compliant since 2001. [50]

2.2 Architecture

Application's architecture refers to the high-level structure of the components, their arrangement, and the way they communicate.

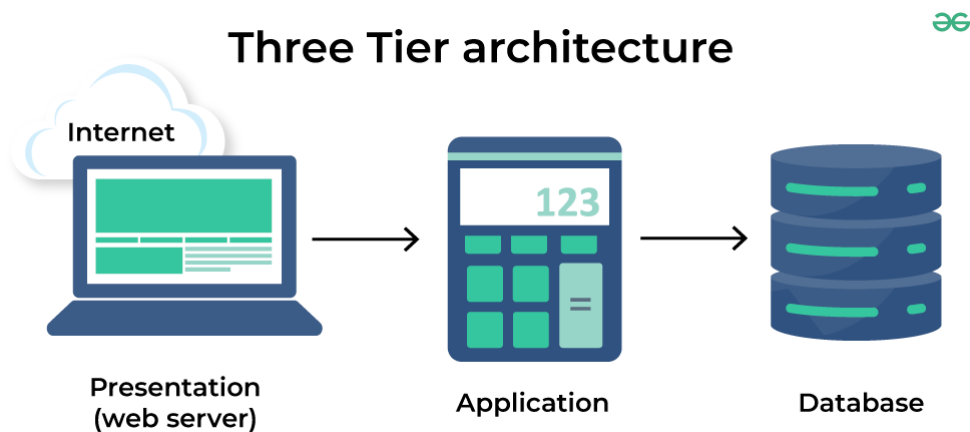
There are several different types of architectures. One of the most commonly used architecture styles is a three-tier client-server architecture, and this kind of architecture is used in the application.

The three-tier architecture is a software application architecture that organizes applications into three logical and physical computing tiers: the presentation tier (or user interface), the application tier (where data is processed), and the data tier, where the data associated with the application are stored and managed. [51]

This kind of architecture has a set of benefits, which include:

- **Logical and physical separation of functionality.** Each tier can run on a separate operating system and server platform, so the services of each tier can be customized and optimized without impacting the other tiers.
- **Better scalability.** Any tier can be scaled independently of the others as needed.
- **Better reliability.** An outage in one tier is less likely to impact the availability or performance of the other tiers.
- **Faster development.** Because each tier can be developed simultaneously by different people, an application can be developed faster, and programmers can use the latest and most suitable technologies for each tier.

Having that said, this section is split into 3 parts: presentation tier, application tier, and data tier.



■ Figure 2.1 Three tier architecture [52]

2.2.1 Presentation tier

The presentation tier is the user interface of the application, where the end user interacts with the application. Its main purpose is to display information and collect information from the user. [51]

The application is designed to support different clients. They can communicate with the server via HTTP requests. In the current scope, only the web client is implemented.

2.2.2 Application tier

The application tier contains the business logic of the application. This tier is responsible for performing computations, data transformations, and other processing tasks. It communicates with the presentation tier, receives and processes requests from it, and sends back the appropriate responses. The application tier also interacts with the data tier, so it can add, delete, or modify data in the data tier. [51]

In case of this application, the application tier is capable of doing the following things:

- Collect users' playbacks.
- Compute users' statistics based on playbacks.
- Compare statistics of different users (e.g. compute their compatibility).
- Other common interactions with the data tier.

It is also designed in a way that new music streaming services can be easily added. Having that said, the resulting application is extensible and sustainable.

2.2.3 Data tier

The data tier is where the information processed by the application is stored and managed. In a three-tier application, all communication goes through the application tier. The presentation tier and the data tier cannot communicate directly with each other. [51]

The database model of the application is presented in Figure 2.2, however, some tables of the database require additional comments:

Stats table represents statistics computed between its **start_date** and **end_date**.

Stats_record contains all computed statistics for a particular user. It includes all-time statistics and a list of temporary statistics (e.g. all weekly, monthly, and yearly records).

Stats_record_recorded_stats is a list of temporary statistics of **Stats_record**.

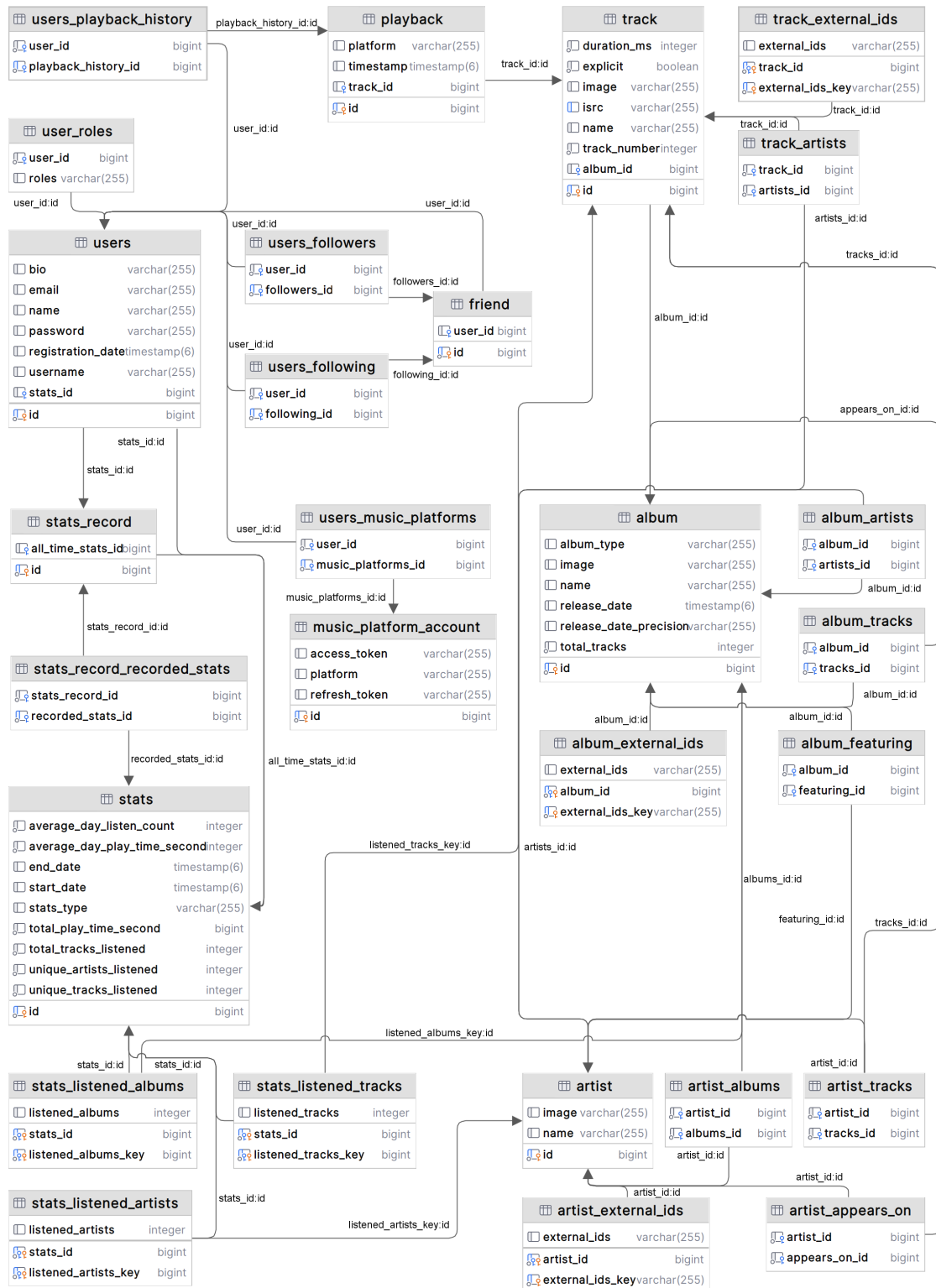
Album_featuring is a list of artists that appear on the album in addition to the main artists on the album.

Artist_appears_on is a list of albums in which the artist features one or more tracks.

2.3 API

An API, or Application Programming Interface, is a set of rules that define how applications or devices can connect to and communicate with each other. A REST API is an API that conforms to the design principles of the REST, or REpresentational State Transfer architectural style. [53]

In total, there are 6 most basic REST design principles. They include a uniform interface, client-server decoupling, statelessness, cacheability, layered system architecture, and code on demand. These principles are outlined in Table 2.1. [53]



■ Figure 2.2 Database model

Principle	Description
Uniform interface	All API requests for the same resource should look the same, no matter where the request comes from. The REST API should ensure that the same piece of data belongs to a single uniform resource identifier (URI).
Client-server decoupling	Client and server applications must be completely independent of each other. The only information that the client application should know is the URI of the requested resource.
Statelessness	REST APIs are stateless, meaning that each request needs to include all the information necessary for processing it.
Cacheability	When possible, resources should be cacheable on the client or server side. Server responses also need to contain information about whether caching is allowed for the delivered resource.
Layered system architecture	REST APIs need to be designed so that neither the client nor the server can tell whether it communicates directly with the end application or with an intermediary.
Code on demand	Optionally, responses can contain executable code on-demand.

■ **Table 2.1** REST principles [53]

The application is designed in a way to comply with these guidelines, therefore it is a RESTful application.

2.3.1 Endpoints

This section describes the most used API endpoints of the application, as well as some specific ones. These endpoints are divided into the following categories: basic entities, compatibility, music services, playbacks, stats, lyrics, authentication, and exceptions.

2.3.1.1 Basic Entities

This section combines common endpoints that just return the required object based on a given identifier.

`/api/albums?id={id}` – returns an album with the requested id.

`/api/artists?id={id}` – returns an artist with the requested id.

`/api/tracks?id={id}` – returns a track with the requested id.

`/api/user?username={username}` – returns a user with the requested username.

2.3.1.2 Compatibility

Compatibility controller provides only one endpoint for getting compatibility between 2 users.

`/api/compatibility?userId={userId}` – returns compatibility between a currently authorized user and a user with the specified userId.

2.3.1.3 Music Services

This category describes endpoints that are used to connect music streaming platform accounts.

`/api/music-services/auth/connect?platform={platform}` – connects specified music streaming service to authenticated user's known services.

`/api/music-services/auth/callback?platform={p}&code={c}` – used for exchanging user authorization code for access and refresh token on a given platform.

It is worth noting that platform can only obtain values listed in **MusicPlatformEnum**. As for now, it includes the following values: "SPOTIFY", "APPLE_MUSIC". However, Apple Music support is not yet implemented.

2.3.1.4 Playbacks

The playback controller operates with user's listening activity on streaming platforms. It provides 2 endpoints: get recently played tracks endpoint and get currently playing track endpoint.

`/api/playback/user-recently-played?userId={userId}&limit={limit}` – returns a list of recently played tracks for a user with the specified id. An optional limit parameter can be applied and its default value is set to 10.

`/api/playback/currently-playing?userId={userId}` – returns currently played track for a user with the specified id.

2.3.1.5 Stats

The stats controller has only one endpoint for getting stats in a specified time range.

`/api/stats?type={type}&userId={id}&objectsLimit={limit}` – returns the most frequent statistics for a user with the specified id. Type parameter defines a time range for requested statistics, and optional parameter objectsLimit sets a number of top albums, artists, and tracks to be returned (default is 5).

Type can only obtain values listed in **StatsTypeEnum**. This enum contains the following values: "WEEKLY", "MONTHLY", "YEARLY", "ALL_TIME".

2.3.1.6 Lyrics

This category is dedicated to the get lyrics endpoint. It requires an ISRC code to be executed and then queries Musixmatch API to fetch track's lyrics [54]. Its free plan allows 2000 API calls per day and returns 30% of the track's lyrics, which should be enough for the showcase of this functionality [55].

`/api/tracks/lyrics?isrc={isrc}` – returns lyrics for a track with the specified ISRC code.

2.3.1.7 Authentication

This section provides an overview of the endpoints that are used for user authentication. They include the register endpoint, the log in endpoint, and the refresh access token endpoint.

/api/auth/register – returns an **AuthenticationResponse** object that contains a user id, username, access and refresh tokens. To be executed, this endpoint requires **RegisterRequest** object with the following fields: email, password, name, and username.

/api/auth/login – returns an **AuthenticationResponse** object. To be executed, this endpoint requires **AuthenticationRequest** object that should contain an email and a password of a registered user.

/api/auth/refresh-access-token – returns a **TokenResponse** object that contains an updated access token and a refresh token. To be executed, it requires a **RefreshRequest** object that must have a refresh token field.

2.3.1.8 Exceptions

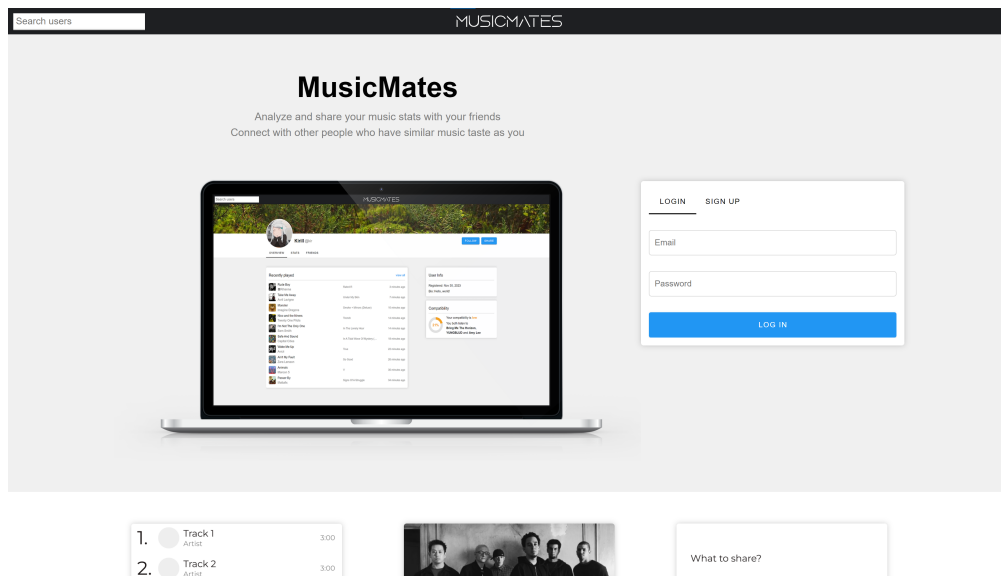
All endpoints may, in some cases, return an exception. These exceptions have a generalized structure and have the following fields: status, error, message, and timestamp. Status is the exception's status code number, the error field describes the reason for the exception, the message field is shown to the user, and a timestamp indicates when this exception has occurred.

2.4 Graphical User Interface

Graphical User Interface is an important part of every web application. It is a visual interface that allows users to interact with the application through graphical elements such as icons, buttons, tabs, etc. This section contains hi-fi prototypes of the design of the application's pages [56].

2.4.1 Main page

Main page is the first thing users see when they enter a website. In case of this application, the main page briefly describes the functionality and contains a form to log in and sign up.



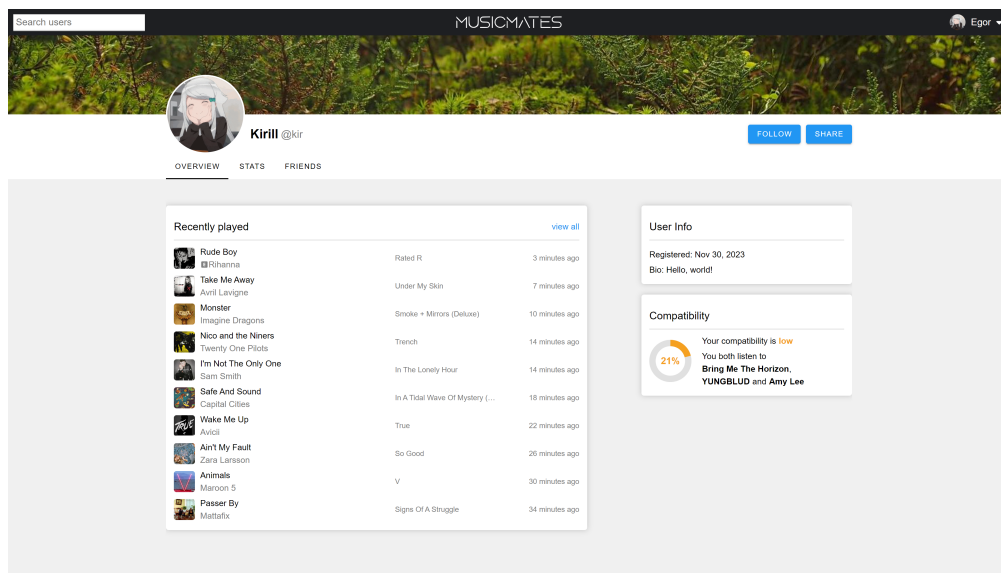
■ Figure 2.3 Main page

2.4.2 User profile

User profile contains data about the user. This is the page user sees after logging in. It has 3 tabs that are used for switching between displayed components: an overview, stats, and friends. Profile header with user's name, username, and profile picture remains unchanged for each of these tabs, however, the majority of the content changes depending on the currently active tab. Header also includes buttons to follow or unfollow a user, edit a profile, and share user's page (by copying its URL to the clipboard).

2.4.2.1 Overview tab

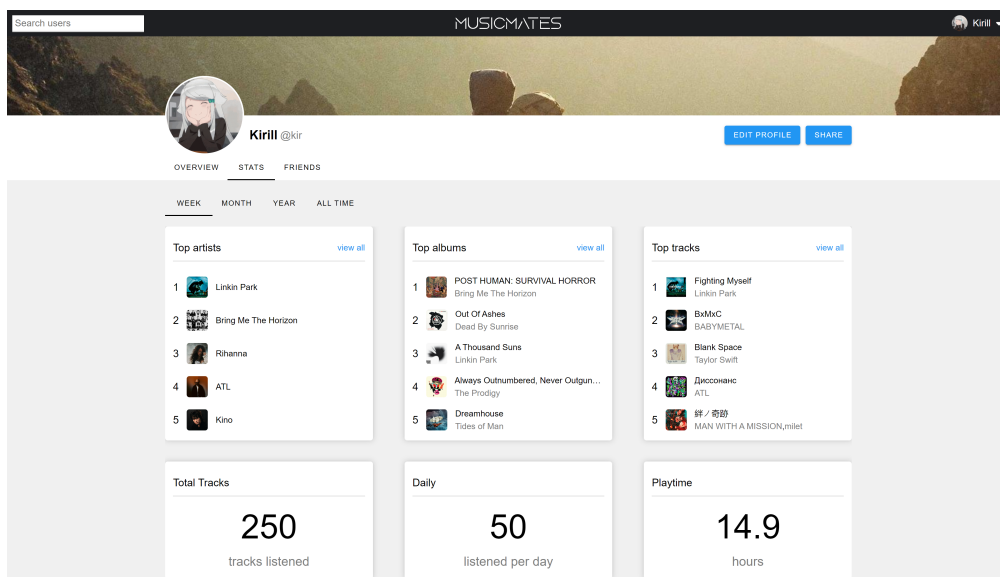
Overview is the main tab in the user's profile. It contains a list of recently played tracks, some basic information about the user, and computed compatibility with the user (if it's applicable). Compatibility includes a score in the 0-100 range and top 3 common artists.



■ Figure 2.4 Overview tab

2.4.2.2 Stats tab

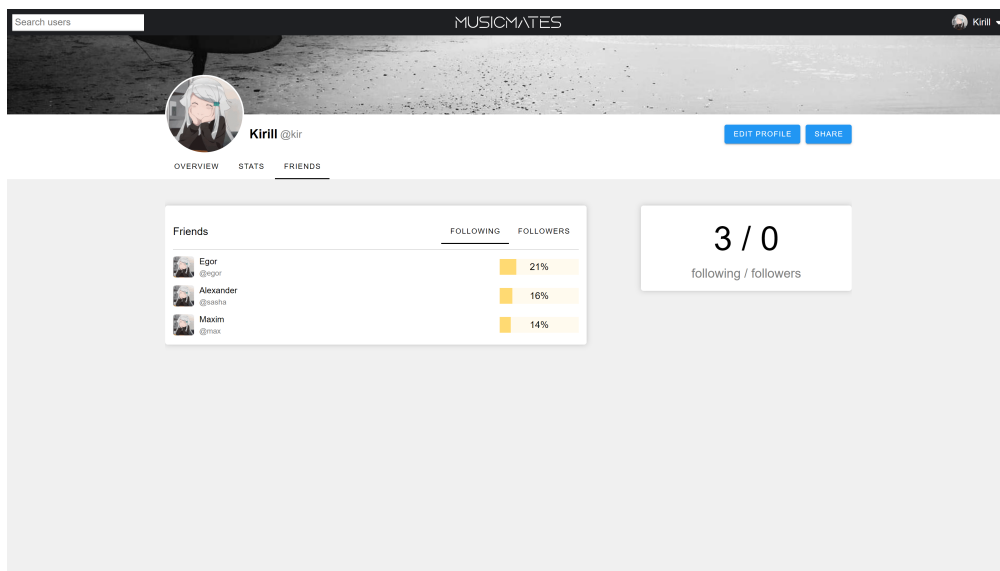
Statistics tab contains data on the user's listening activity. It stores information about user's most listened albums, artists, and tracks. Clicking the "view all" button opens a modal window that contains a larger list of top entities of the corresponding type, as well as the number of times user has listened to it. Besides that, this page also contains the overall number of tracks listened in the current time period, amount of tracks listened daily, and a total amount of time spent listening to music.



■ Figure 2.5 Stats tab

2.4.2.3 Friends tab

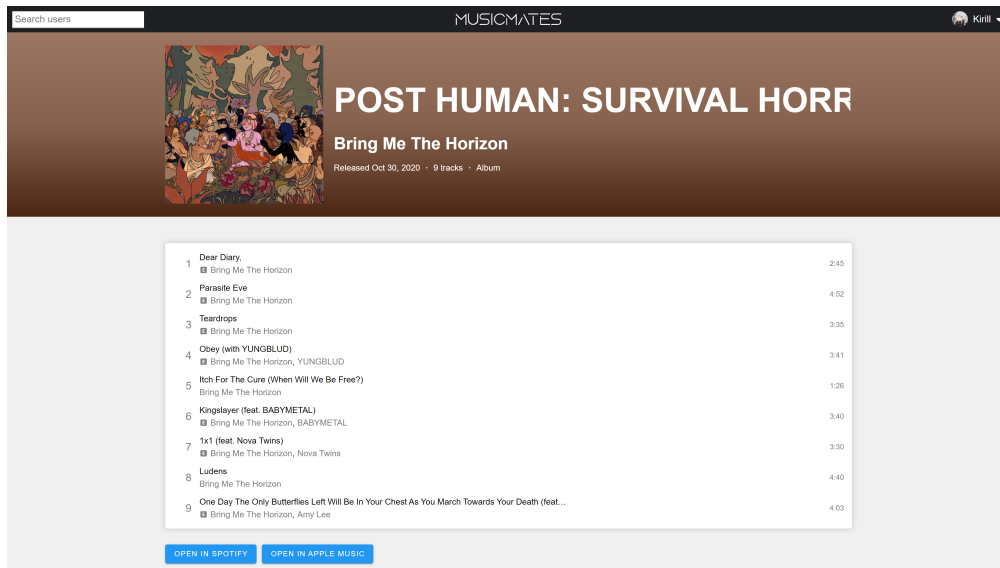
Friends tab contains a list of users that the current user follows and a list of users that follow the current user. Compatibility of musical preferences is also shown for each displayed user, and they are sorted based on their compatibility.



■ Figure 2.6 Friends tab

2.4.3 Album page

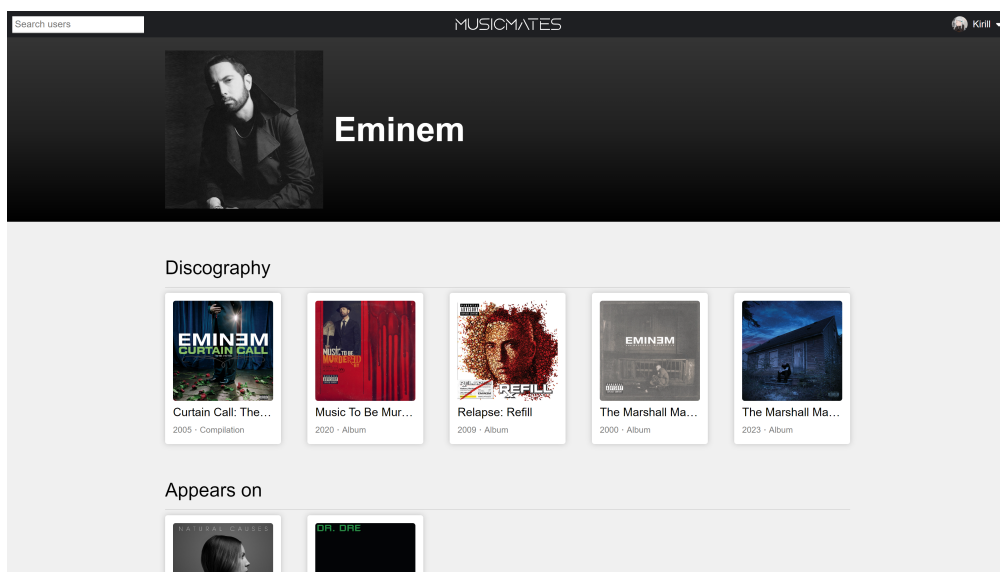
Album page displays the information about the album. It shows the name of the album, its image, artists and metadata, a list of tracks on this album, and links to this album on supported music platforms.



■ Figure 2.7 Album page

2.4.4 Artist page

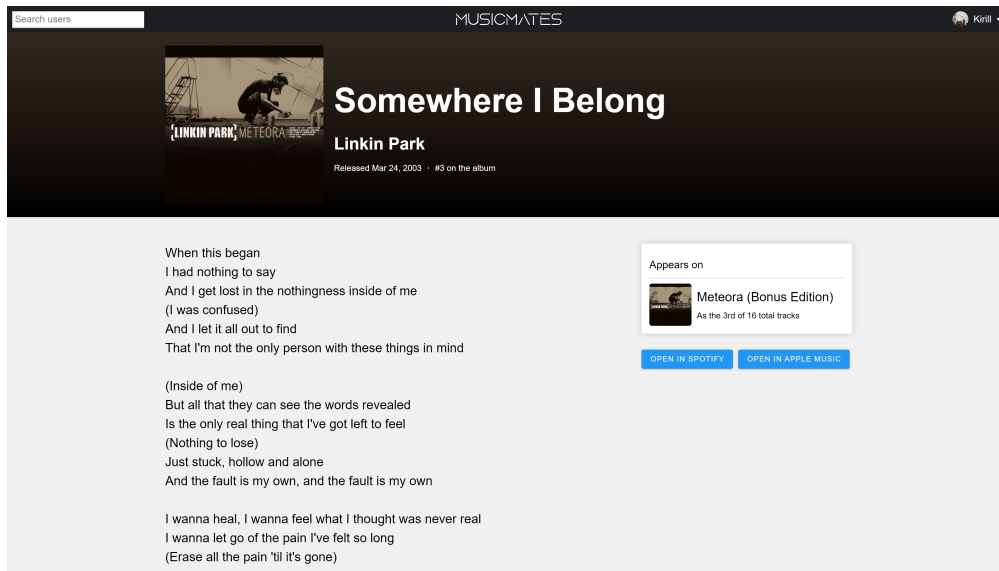
Artist page contains basic data about the artist. It includes artist's name and image, a list of artist's albums, and a list of albums this artist appears on. Also, this page has links to the artist page on music platforms.



■ Figure 2.8 Artist page

2.4.5 Track page

Track page presents data about the track. This data includes track's name and artists, its metadata, its number on the album, and album's image. It also contains a part of lyrics fetched from Musixmatch API (described in Section 2.3.1.6). Additionally, it has links to this track on corresponding music streaming services.



The screenshot shows the track page for "Somewhere I Belong" by Linkin Park on the MusicMates website. The page features a dark header with a search bar, the site name "MUSICMATES", and a user profile "Kriil". The track title "Somewhere I Belong" is prominently displayed in white, with the artist "Linkin Park" below it. A small album cover for "METEORA" is shown to the left. Below the track information, the lyrics are displayed in a light gray font. On the right side, there is a section titled "Appears on" which shows the album "Metora (Bonus Edition)" and indicates it is the 3rd of 16 total tracks. Two buttons, "OPEN IN SPOTIFY" and "OPEN IN APPLE MUSIC", are located below this section.

Search users MUSICMATES Kriil

Somewhere I Belong
Linkin Park
Released Mar 24, 2003 · #3 on the album

When this began
I had nothing to say
And I get lost in the nothingness inside of me
(I was confused)
And I let it all out to find
That I'm not the only person with these things in mind

(Inside of me)
But all that they can see the words revealed
Is the only real thing that I've got left to feel
(Nothing to lose)
Just stuck, hollow and alone
And the fault is my own, and the fault is my own

I wanna heal, I wanna feel what I thought was never real
I wanna let go of the pain I've felt so long
(Erase all the pain 'til it's gone)
I wanna heal, I wanna feel like I'm close to something real

Appears on
Metora (Bonus Edition)
As the 3rd of 16 total tracks

OPEN IN SPOTIFY OPEN IN APPLE MUSIC

■ Figure 2.9 Track page

Implementation

This chapter is devoted to the implementation details of the application. It includes code snippets of the most noteworthy parts on both the server and the client side of the application. Apart from that, it contains their explanations and reasonings behind the decisions made.

3.1 Backend implementation

This section describes some of the features implemented on the backend side of the application. It includes computing statistics, updating playbacks, mapping albums, artists, and tracks between different music streaming services, calculating compatibility between users, fetching lyrics, and the security of the application.

3.1.1 Computing statistics

Computing statistics is one of the "must have" requirements set in Section 1.3.1. There are 4 types of statistics in the application: weekly, monthly, yearly, and all-time. All-time statistics are updated after each recorded playback to ensure that presented statistics are accurate and up to date. However, other types of statistics are calculated differently. Weekly statistics are generated each Friday using a scheduled method. Friday was deliberately chosen because most of the tracks are released on Friday [57]. Monthly statistics are generated at the beginning of each month, and yearly statistics are generated on the 1st of January. Monthly and yearly statistics reuse already computed data by weekly and monthly statistics, respectively.

3.1.2 Updating playbacks

The application generates statistics based on users' listening activity. It fetches recently played tracks and stores them in the database. Doing that allows to compute statistics in any desired way without relying on statistics calculated by music platforms (moreover not all of them expose user statistics). However, endpoints that allow to get recently played tracks return a limited amount of tracks. In case of Spotify, it can return only the last 50 tracks. So to make statistics precise it is necessary to periodically update users' playback history to ensure that all listened tracks are recorded in the database.

To accomplish this, a scheduled method with a fixed delay was implemented. It fetches playbacks for all users once an hour from the streaming services they have linked. It can be done in the background, and no actions are required from the users because their access and refresh tokens for music platforms are stored in the database.

```

@Transactional
@Scheduled(fixedDelay = 1000 * 60 * 60)
override fun updateUsersPlaybacks() {
    userRepository.findAll().forEach { user ->
        fetchNewPlaybacks(user)
    }
}

override fun fetchNewPlaybacks(user: User) {
    val newPlaybacks = mutableListOf<PlaybackDTO>()
    musicPlatformApiServices.forEach { (platform, platformApiService) ->
        if (musicPlatformAccountService.userHasAccount(user, platform)) {
            newPlaybacks.addAll(platformApiService
                .getNewPlaybacks(user, calcFetchAfter(user)))
        }
    }
    newPlaybacks.sortedBy { it.timestamp }.forEach {
        playbackService.createPlayback(user, it)
    }
}

private fun calcFetchAfter(user: User) = user.playbackHistory
    .maxByOrNull { it.timestamp }?.timestamp

```

■ Code listing 1 Updating playbacks

Listing 1 presents 3 functions: *updateUsersPlaybacks*, *fetchNewPlaybacks*, and *calcFetchAfter*. The first function triggers fetching new playbacks for each registered user once an hour. The second function retrieves playbacks for a given user from all music streaming services that are linked to the user's account. After retrieving, it calls the *createPlayback* function that saves all listened track records to the database. The third function returns the timestamp of the last listened track. It is used to fetch only new tracks from music platforms.

3.1.3 Mapping platform objects

As was previously stated in Section 1.3.1, it is important for the application to support multiple music streaming services. Doing that will increase the amount of potential users and allow to fetch data about the albums, artists, and tracks from multiple sources. Despite the fact that in the current state only Spotify is supported, the application is designed to support more streaming services.

In order to map albums, artists, and tracks between different music platforms, a set of actions have been done. It was already mentioned in Section 1.1.1.5 that Spotify exposes ISRC codes of the tracks. This code is a unique identifier of a record, and other popular streaming services like Apple Music and Tidal allow to query tracks by their ISRC [10, 58]. That said, tracks can be mapped between these platforms. When the track is created, the application queries all supported platforms to get a track by its ISRC. Track entity also includes an album of the track and artists track belongs to. Their presence in the track entity solves the problem of mapping albums and artists as they do not have unique identifiers exposed by music streaming services.

Listing 2 demonstrates how album, artists, and track identifiers are fetched from all supported music platforms by track's ISRC.


```

override fun fillPlatformIds(track: TrackDTO): TrackDTO {
    musicPlatformApiServices.forEach { (platform, platformApiService) ->
        // Skip if track already contains platform-specific ids
        if (track.externalIds.containsKey(platform)) return@forEach

        // Object with album, artists, and track platform-specific ids
        val idsDto = platformApiService.getPlatformIdsByIsrc(track.isrc)
        track.externalIds[platform] = idsDto.trackId
        track.artists.forEachIndexed { index, artist ->
            artist.externalIds[platform] = idsDto.trackArtistIds[index]
        }
        track.album.externalIds[platform] = idsDto.albumId
        track.album.artists.forEachIndexed { index, artist ->
            artist.externalIds[platform] = idsDto.albumArtistIds[index]
        }
    }
    return track
}

```

■ **Code listing 2** Mapping objects

3.1.4 Calculating compatibility

Compatibility calculation is one of the required features of the application. At this moment, it is possible to calculate compatibility between 2 users based on their all-time top artists.

3.1.5 Security

This application uses JWT tokens for authorizing requests. To act on behalf of a user, a request must include a JWT token in its authorization header. When the server receives the request, it checks the validity of the JWT token and then grants access to desired resource.

3.1.6 Lyrics fetching

As was already mentioned in Section 2.3.1.6 and Section 2.4.5, the application uses Musixmatch to fetch tracks' lyrics. Initially, it fetches the track by its ISRC code to obtain the track's Musixmatch identifier, and once it is acquired, it becomes possible to fetch lyrics [59, 60].

```

override fun getLyrics(isrc: String): LyricsDTO {
    val trackJson = getMusixmatchTrackByIsrc(isrc)
    if (!trackHasLyrics(trackJson)) return LyricsDTO()
    val trackId = extractTrackId(trackJson)
    val lyricsJson = getLyricsByMusicmatchTrackId(trackId)
    ?: return LyricsDTO()
    return LyricsDTO(extractLanguage(lyricsJson), extractText(lyricsJson))
}

```

■ **Code listing 3** Lyrics fetching

3.2 Frontend implementation

This section presents code snippets of the frontend part of the application. Apart from that, it contains a brief description of composable functions in Vue and the Tanstack Query library that is used in the application.

3.2.1 Composables

In the context of Vue applications, a "composable" is a function that leverages Vue's Composition API to encapsulate and reuse stateful logic. These functions take some input and immediately return the expected output. [61]

Composable functions solve a variety of problems. In this application, they convert a date to a string, calculate how much time has passed since some specified moment, transform track's duration from milliseconds to a string, etc.

```
export function toDateString(date: Date): string {
  const dateString = date.toString().split("T")[0];
  const [year, month, day] = dateString.split("-");
  return `${monthConverter(month)} ${day}, ${year}`;
}
```

■ **Code listing 4** Date to string composable

```
export function durationToString(durationMs: number): string {
  const minutes = Math.floor(durationMs / 60000);
  const seconds = +(((durationMs % 60000) / 1000).toFixed(0));
  return `${minutes}:${seconds < 10 ? "0" : ""}${seconds}`;
}
```

■ **Code listing 5** Duration to string composable

Listing 4 and Listing 5 show some composable functions that are implemented in the application.

3.2.2 TanStack Query

TanStack Query is a library that provides asynchronous state management for TS/JS, and for frontend frameworks like React, Solid, Vue, Svelte, and Angular. This library simplifies the process of fetching, caching, synchronizing, and updating server state in web applications. [62, 63]

The implemented application leverages some features of the TanStack Query library, such as setting an amount of retries on request sending, refetching data on window focus after a specified period of time, and so on.

```

export const useRequest = <T extends any>(params: QueryParams<T>):
UseQueryReturnType<T, any> => {
  return useQuery({
    queryKey: params.queryKey,
    queryFn: async () => {
      const response = await callAPI('GET', params.url)
      if (response.status === 204) return null
      return await response.json() as T
    },
    retry: params.retry ?? 3,
    refetchOnWindowFocus: params.refetch,
    refetchOnReconnect: params.refetch,
    staleTime: params.refetch ? 1000 * 60 : 1000 * 60 * 60,
    enabled: params.enabled
  })
}

export const callAPI = async (method: string, url: string, data?: any) => {
  return fetch(`${BaseURL}${url}`, {
    method: method,
    headers: getHeaders(),
    body: JSON.stringify(data)
  })
}

```

■ Code listing 6 Tanstack Query wrapping

Listing 6 demonstrates how TanStack's *useQuery* method is wrapped in the application's codebase. It presents a *useRequest* function that takes the *QueryParams* object as its parameter. Based on these query parameters, corresponding fields of the *useQuery* method are set. This snippet also shows an asynchronous method *callAPI* whose purpose is to send a request to a given URL.


```

export function getPlaybacks(userId: number, limit: number):
UseQueryReturnType<Playback[], any> {
  return useRequest<Playback[]>({
    url: `/playback/user-recently-played?userId=${userId}&limit=${limit}`,
    queryKey: ['playback', userId.toString(), limit.toString()],
    refetch: true,
  })
}

```

■ Code listing 7 Get playbacks function

Listing 7 demonstrates how recently played tracks are fetched. The presented *getPlaybacks* method calls the *useRequest* function with the *refetch* parameter set to true. This way, recently played tracks are periodically refetched from the server, but only when a user is viewing the window of the application.



Chapter 4

Testing

In order to make further development of the application easier and to maintain its sustainability and reliability, it is necessary to cover the application with proper tests. This chapter describes tests that were performed on the application and outlines their results.

4.1 Types of tests

There are numerous types of tests that are used in applications development nowadays. They include unit tests, integration tests, end-to-end tests, smoke tests, etc. In the current scope, the application was tested with unit tests and manual tests performed by invited testers.

4.1.1 Unit testing

Unit tests are used to test small components of the application. The purpose of unit testing is to validate that each unit of the software code performs as expected. Unit tests isolate a section of code and verify its correctness. [64]

In this application, unit tests cover individual methods of services on the backend side. They check whether these methods work as intended in isolation by mocking external dependencies. This application uses the MockK library [65] for this purpose.

4.1.2 Manual testing

Manual testing is a type of software testing that is performed by users who test the application according to predefined scenarios without using any automated tools. These tests are meant to simulate the usage of the application, and their purpose is to identify bugs, issues, and defects in the software application. [66]

Manual testing of the application was performed by 5 users. Since the application is still in development mode and access to Spotify API in this mode is restricted, these 5 users were added to the list of authorized users in the Spotify Developer Dashboard [67]. To make manual testing by other users possible, the application had to be deployed. That said, the frontend part was deployed on Vercel [68] while the database and the server part were deployed to Railway [69]. These solutions were chosen due to their speed, reliability, and straightforwardness.

4.2 Results of tests

Unit tests increased the reliability of the application. They check the correctness of code changes and assist with spotting potential bugs.

As for the manual tests carried out by other users, they went according to expectations. There were no complaints regarding the core functionality of the application. Minor spotted bugs were fixed, and user experience was improved in accordance with the feedback from the testers. In addition, some ideas on further application improvement were considered valuable and were added to Section 5.3.

Evaluation

This chapter contains an overall evaluation of the resulting application. It covers the usability of the application, compares it with already existing solutions examined in the analysis chapter, and outlines possible future improvements.

5.1 Usability

Despite the fact that this is the first version of the application, all necessary features for conducting an analysis of musical preferences and comparing them with other users are implemented. In the current state, the application fulfills all the "must have" and "should have" requirements set in Section 1.3. That said, it can be considered usable, and the feedback gained from testers confirms it.

5.2 Comparison with existing solutions

As mentioned in Section 1.2, currently only one solution supports multiple music streaming services. Even though the current version of the application only supports Spotify, it is designed in a way that allows to easily connect other streaming services to it. Additionally, the application has its own way of computing statistics based on the listening history and supports comparing musical preferences. These points make the resulting application a viable competitor to already existing solutions.

5.3 Future improvements

Although the resulting application meets all set requirements, it is important to set goals for further development. This section outlines potential features for future implementation that emerged while working on the project, as well as ideas gained from testers' feedback.

5.3.1 Extended statistics

One of the possible improvements is to provide users with more statistics on their listening activity. Enough statistics are being collected and stored even now, however, not all of them are yet shown to the end user. Also, storing weekly, monthly, and yearly statistics snapshots allows to display trends in the listening activity of the user.

5.3.2 Profound preferences comparison

Similarly to statistics, more profound preferences comparison can be shown to the user. Currently, only a score and a top 3 common artists are presented to the user, despite the fact that more data are collected and used for computing users' compatibility rate. Apart from that, the compatibility score can depend not only on the number of users' common artists but also on their ordering and other parameters like top albums, tracks, and genres.

5.3.3 Support more music services

The application is designed in a way to support multiple music streaming services. This opportunity should be leveraged in the future as it will allow to gain more data about albums, artists, and tracks, and potentially will bring more users to the application.

5.3.4 Responsive design

As users have different resolutions and aspect ratios of their displays, it is necessary to make the client part of the application responsive. It means that the application should be able to adapt to different devices, resolutions, and screen sizes to ensure a good user experience.

5.3.5 Custom profile images

In the current state, the application does not support uploading custom profile and cover images. It was not a necessity in the current scope of development, however, it should be implemented in the future.

5.3.6 Complete album fetching

Right now, when a user listens to a track that does not exist in the application's database, this track is added to the database. Moreover, if the album of this track does not exist, it is created as well. However, with this approach, albums are incomplete until all their tracks have been listened to by application's users. One possible solution can be to fetch all album's tracks when the album has to be created.

5.3.7 Mobile applications

A lot of people use smartphones on a daily basis, therefore implementing mobile applications for iOS and Android may greatly broaden the audience of the application. Moreover, it will provide a more pleasant experience to already registered users.

Conclusion

The goal of this thesis was to design, implement, and test a web application that would allow its users to examine their musical preferences and compare them with other users.

The first chapter of the thesis was dedicated to the research of the Spotify Web API, analysis of already existing solutions, and setting functional and non-functional requirements for the application following the MoSCoW principle.

The second chapter provided an overview of the design of the desired software. It briefly described technologies that are used in the application and the overall architecture of the application, including its API. This chapter also provided a showcase of the application's user interface.

The implementation chapter presented code snippets of the most noteworthy parts of the application, accompanied by the corresponding descriptions of their functionality.

The fourth chapter was dedicated to the testing of the application. It described the types of tests that were performed during development and recapped their results.

The evaluation chapter proved the usability of the application, compared the resulting software with already existing solutions, and outlined possible future improvements. These improvements include more profound statistics generation and preferences comparison, supporting other music streaming services, making the UI more responsive, supporting custom profile images, complete album fetching, and developing mobile applications to broaden the application's audience.

Having that said, the thesis successfully fulfilled all stated goals. Research on the Spotify API and existing similar solutions was conducted, application was designed and implemented following set functional and non-functional requirements and according to best practices of software engineering. It was properly tested and evaluated, and potential future improvements were presented.

Bibliography

1. *Web API* [online]. Spotify, 2023 [visited on 2023-12-09]. Available from: <https://developer.spotify.com/documentation/web-api>.
2. *API calls* [online]. Spotify, 2023 [visited on 2023-12-09]. Available from: <https://developer.spotify.com/documentation/web-api/concepts/api-calls>.
3. *Get Album* [online]. Spotify, 2023 [visited on 2023-12-09]. Available from: <https://developer.spotify.com/documentation/web-api/reference/get-an-album>.
4. *Get Several Albums* [online]. Spotify, 2023 [visited on 2023-12-09]. Available from: <https://developer.spotify.com/documentation/web-api/reference/get-multiple-albums>.
5. *Get Artist* [online]. Spotify, 2023 [visited on 2023-12-09]. Available from: <https://developer.spotify.com/documentation/web-api/reference/get-an-artist>.
6. *Get Several Artists* [online]. Spotify, 2023 [visited on 2023-12-09]. Available from: <https://developer.spotify.com/documentation/web-api/reference/get-multiple-artists>.
7. *Get Track* [online]. Spotify, 2023 [visited on 2023-12-09]. Available from: <https://developer.spotify.com/documentation/web-api/reference/get-track>.
8. *International Standard Recording Code* [online]. Wikipedia, 2023 [visited on 2023-12-09]. Available from: https://en.wikipedia.org/wiki/International_Standard_Recording_Code.
9. *Songs Attributes* [online]. Apple Inc., 2023 [visited on 2023-12-09]. Available from: <https://developer.apple.com/documentation/applemusicapi/songs/attributes>.
10. *Get Multiple Catalog Songs by ISRC* [online]. Apple Inc., 2023 [visited on 2023-12-09]. Available from: https://developer.apple.com/documentation/applemusicapi/get_multiple_catalog_songs_by_isrc.
11. *Get Several Tracks* [online]. Spotify, 2023 [visited on 2023-12-09]. Available from: <https://developer.spotify.com/documentation/web-api/reference/get-several-tracks>.
12. *Search for Item* [online]. Spotify, 2023 [visited on 2023-12-09]. Available from: <https://developer.spotify.com/documentation/web-api/reference/search>.
13. *Get User's Top Items* [online]. Spotify, 2023 [visited on 2023-12-09]. Available from: <https://developer.spotify.com/documentation/web-api/reference/get-users-top-artists-and-tracks>.
14. *Get Recently Played Tracks* [online]. Spotify, 2023 [visited on 2023-12-09]. Available from: <https://developer.spotify.com/documentation/web-api/reference/get-recently-played>.

15. *Get Currently Playing Track* [online]. Spotify, 2023 [visited on 2023-12-09]. Available from: <https://developer.spotify.com/documentation/web-api/reference/get-the-users-currently-playing-track>.
16. *Get Track's Audio Features* [online]. Spotify, 2023 [visited on 2023-12-09]. Available from: <https://developer.spotify.com/documentation/web-api/reference/get-audio-features>.
17. *Last.fm* [online]. Last.fm, 2023 [visited on 2023-12-09]. Available from: <https://last.fm>.
18. *About Last.fm* [online]. Last.fm, 2023 [visited on 2023-12-09]. Available from: <https://last.fm/about>.
19. *What is a Scrobble and what is Scrobbling?* [online]. Last.fm, 2015 [visited on 2023-12-09]. Available from: https://cbsi.my.salesforce-sites.com/lastfm/articles/en_US/Knowledge/What-is-scrobbling?template=template_lastfm&referrer=lastfm.com.
20. *Track My Music* [online]. Last.fm, 2023 [visited on 2023-12-09]. Available from: <https://www.last.fm/about/trackmymusic>.
21. *More ways to Scrobble* [online]. Last.fm, 2019 [visited on 2023-12-09]. Available from: <https://support.last.fm/t/more-ways-to-scrobble/192>.
22. *stats.fm* [online]. stats.fm, 2023 [visited on 2023-12-09]. Available from: <https://stats.fm>.
23. *stats.fm Plus* [online]. stats.fm, 2023 [visited on 2023-12-09]. Available from: <https://stats.fm/plus>.
24. *Differences between calculation methods* [online]. stats.fm, 2023 [visited on 2023-12-09]. Available from: <https://support.stats.fm/docs/import/faq/calculation-methods>.
25. *Streaming history synchronisation* [online]. stats.fm, 2023 [visited on 2023-12-09]. Available from: <https://support.stats.fm/docs/streams/sync>.
26. *Stats for Spotify* [online]. Stats for Spotify, 2023 [visited on 2023-12-09]. Available from: <https://www.statsforspotify.com/>.
27. *Obscurify* [online]. Obscurify, 2023 [visited on 2023-12-09]. Available from: <https://www.obscurifymusic.com/>.
28. *About Obscurify* [online]. Obscurify, 2023 [visited on 2023-12-09]. Available from: <https://www.obscurifymusic.com/about>.
29. *musictaste.space* [online]. musictaste.space, 2023 [visited on 2023-12-09]. Available from: <https://musictaste.space/>.
30. *About musictaste.space* [online]. musictaste.space, 2023 [visited on 2023-12-09]. Available from: <https://musictaste.space/about>.
31. *MoSCoW method* [online]. TechTarget, 2023 [visited on 2023-12-09]. Available from: <https://www.techtarget.com/searchsoftwarequality/definition/MoSCoW-method>.
32. *Functional vs Non Functional Requirements* [online]. Guru99, 2023 [visited on 2023-12-09]. Available from: <https://www.guru99.com/functional-vs-non-functional-requirements.html>.
33. *Frontend vs Backend* [online]. GeeksforGeeks, 2023 [visited on 2024-01-30]. Available from: <https://www.geeksforgeeks.org/frontend-vs-backend/>.
34. *TypeScript* [online]. Microsoft, 2024 [visited on 2024-01-25]. Available from: <https://www.typescriptlang.org/>.
35. *The Progressive JavaScript Framework* [online]. Evan You, 2024 [visited on 2024-01-25]. Available from: <https://vuejs.org/>.
36. *Introduction* [online]. Evan You, 2024 [visited on 2024-01-25]. Available from: <https://vuejs.org/guide/introduction.html>.

37. *Comparison with Other Frameworks* [online]. Evan You, 2024 [visited on 2024-01-25]. Available from: <https://v2.vuejs.org/v2/guide/comparison.html>.
38. *Kotlin* [online]. Kotlin, 2024 [visited on 2024-01-25]. Available from: <https://kotlinlang.org/>.
39. *Kotlin (programming language)* [online]. Wikipedia, 2024 [visited on 2024-01-25]. Available from: [https://en.wikipedia.org/wiki/Kotlin_\(programming_language\)](https://en.wikipedia.org/wiki/Kotlin_(programming_language)).
40. *Spring Framework* [online]. Broadcom, 2024 [visited on 2024-01-25]. Available from: <https://spring.io/projects/spring-framework/>.
41. *Why Spring?* [online]. Broadcom, 2024 [visited on 2024-01-25]. Available from: <https://spring.io/why-spring/>.
42. *Introducing Kotlin support in Spring Framework 5.0* [online]. Broadcom, 2017 [visited on 2024-01-25]. Available from: <https://spring.io/blog/2017/01/04/introducing-kotlin-support-in-spring-framework-5-0/>.
43. *Kotlin for server side* [online]. Kotlin, 2023 [visited on 2024-01-25]. Available from: <https://kotlinlang.org/docs/server-overview.html>.
44. *Spring Boot* [online]. Broadcom, 2024 [visited on 2024-01-25]. Available from: <https://spring.io/projects/spring-boot/>.
45. *Spring Security* [online]. Broadcom, 2024 [visited on 2024-01-25]. Available from: <https://spring.io/projects/spring-security/>.
46. *Gradle* [online]. Gradle Inc., 2024 [visited on 2024-01-25]. Available from: <https://gradle.org/>.
47. *Gradle vs Maven Comparison* [online]. Gradle Inc., 2024 [visited on 2024-01-25]. Available from: <https://gradle.org/maven-vs-gradle/>.
48. *Gradle + Kotlin* [online]. Gradle Inc., 2024 [visited on 2024-01-25]. Available from: <https://gradle.org/kotlin/>.
49. *Relational vs. Non-Relational Databases* [online]. MongoDB, Inc., 2023 [visited on 2024-01-25]. Available from: <https://www.mongodb.com/compare/relational-vs-non-relational-databases>.
50. *About* [online]. The PostgreSQL Global Development Group, 2024 [visited on 2024-01-25]. Available from: <https://www.postgresql.org/about/>.
51. *What is three-tier architecture?* [online]. IBM, 2024 [visited on 2024-01-25]. Available from: <https://www.ibm.com/topics/three-tier-architecture>.
52. *Three-Tier Client Server Architecture in Distributed System* [online]. GeeksforGeeks, 2023 [visited on 2024-01-30]. Available from: <https://www.geeksforgeeks.org/three-tier-client-server-architecture-in-distributed-system/>.
53. *What is a REST API?* [online]. IBM, 2024 [visited on 2024-01-25]. Available from: <https://www.ibm.com/topics/rest-apis>.
54. *Build with Lyrics* [online]. Musixmatch, 2024 [visited on 2024-01-25]. Available from: <https://developer.musixmatch.com/>.
55. *Pricing & Plans* [online]. Musixmatch, 2024 [visited on 2024-01-25]. Available from: <https://developer.musixmatch.com/plans>.
56. *What is high-fidelity prototyping—and how can it help?* [online]. Figma, 2024 [visited on 2024-01-30]. Available from: <https://www.figma.com/resource-library/high-fidelity-prototyping/>.
57. *International Standard Recording Code* [online]. Wikipedia, 2024 [visited on 2024-02-08]. Available from: https://en.wikipedia.org/wiki/Global_Release_Day.

58. *Get tracks by ISRC* [online]. TIDAL Music AS, 2024 [visited on 2024-02-08]. Available from: <https://apiref.developer.tidal.com/apiref?spec=catalogue&ref=get-tracks-by-isrc>.
59. *track.get* [online]. Musixmatch, 2024 [visited on 2024-02-08]. Available from: <https://developer.musixmatch.com/documentation/api-reference/track-get>.
60. *track.lyrics.get* [online]. Musixmatch, 2024 [visited on 2024-02-08]. Available from: <https://developer.musixmatch.com/documentation/api-reference/track-lyrics-get>.
61. *Composables* [online]. Evan You, 2024 [visited on 2024-02-10]. Available from: <https://vuejs.org/guide/reusability/composables>.
62. *TanStack Query* [online]. Tanner Linsley, 2024 [visited on 2024-02-10]. Available from: <https://tanstack.com/query/latest>.
63. *Overview* [online]. Tanner Linsley, 2024 [visited on 2024-02-10]. Available from: <https://tanstack.com/query/latest/docs/framework/vue/overview>.
64. *What is Unit Testing?* [online]. Guru99, 2023 [visited on 2024-01-30]. Available from: <https://www.guru99.com/unit-testing-guide.html>.
65. *MockK* [online]. Tanner Linsley, 2024 [visited on 2024-02-10]. Available from: <https://mockk.io/>.
66. *Manual Testing Tutorial* [online]. Guru99, 2023 [visited on 2024-01-30]. Available from: <https://www.guru99.com/manual-testing.html>.
67. *Quota modes* [online]. Spotify, 2024 [visited on 2024-01-30]. Available from: <https://developer.spotify.com/documentation/web-api/concepts/quota-modes>.
68. *Vercel is the Frontend Cloud.* [online]. Vercel, 2024 [visited on 2024-02-04]. Available from: <https://vercel.com/>.
69. *Instant Deployments, Effortless Scale* [online]. Railway Corp., 2024 [visited on 2024-02-04]. Available from: <https://railway.app/>.

Contents of the attachment

	readme.txt.....	contents of the attachment description
	examples	
	screenshots.....	images demonstrating the application
	videos.....	videos demonstrating the application
	text	
	thesis.pdf.....	thesis text in PDF format
	thesis.zip.....	archive with L ^A T _E X source code of the thesis