Same Rhythm Radar

Group 2, ITCS 4155

SD^2: Software Design Document

The purpose of the software design document (SD^2) is to provide an overview of the system and to help people understand the system. This document can be used by a programmer as a guideline for implementing the design without needing to make significant engineering design decisions and will be provided to the customer as a final deliverable.

1. Project Overview

1.1 - Overview

Same Rhythm Radar is a full-stack application designed to provide users with personalized song recommendations based on their queries. In today's digital age, music streaming platforms offer vast libraries of songs, making it challenging for users to discover new music tailored to their preferences. Same Rhythm Radar aims to address this problem by leveraging artificial intelligence to analyze user queries and suggest relevant songs, thus enhancing the overall music discovery experience.

1.2 - Problem Statement

The modern music landscape is saturated with a wide variety of songs across various genres, making it difficult for users to navigate and discover new music. Existing music streaming platforms provide extensive catalogs, but users often struggle to find songs that match their moods, preferences, or current trends. Additionally, users may find it cumbersome to manually search for songs or rely on generic recommendations.

1.3 - Problem Solution

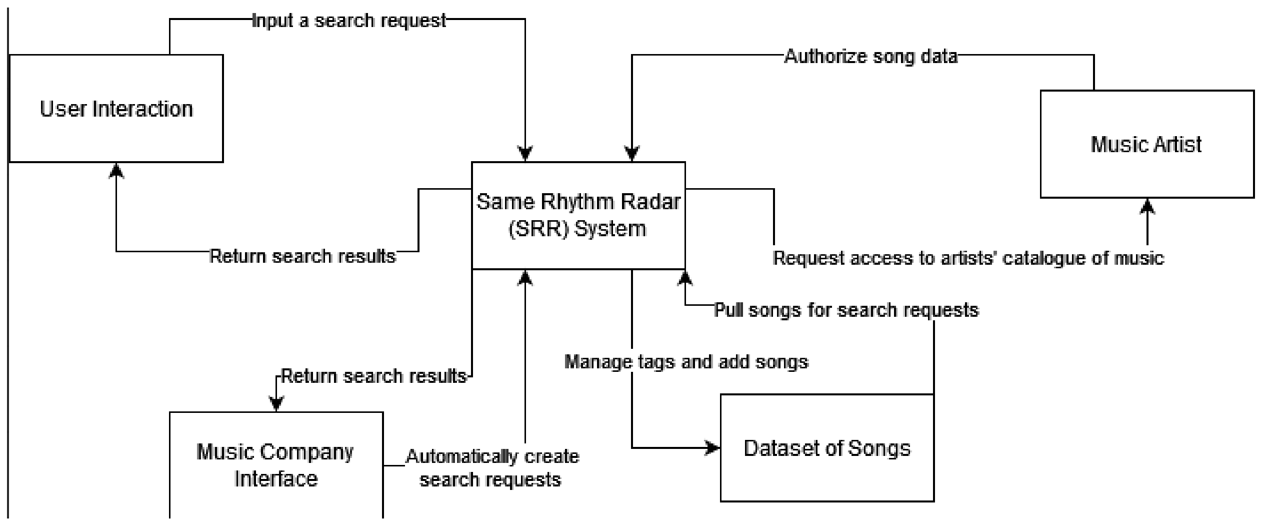
Same Rhythm Radar offers a user-friendly interface where users can input queries in natural language to receive personalized song recommendations. By integrating artificial intelligence algorithms, the application analyzes user queries, considers factors such as genre, mood, popularity, and user history, and generates relevant song suggestions. These suggestions aim to match the user's preferences and provide an engaging music discovery experience.

1.4 - Stakeholders

- **Users:** Music enthusiasts who seek personalized song recommendations based on their preferences and moods.

- Music Streaming Platforms (e.g. Spotify, Apple Music): Companies offering extensive music libraries and seeking to enhance user engagement and satisfaction through improved music discovery features

1.5 - Context Diagram


User Interaction
Input a search request
Authorize song data
Music Artist
Same Rhythm Radar (SRR) System
Return search results
Request access to artists' catalogue of music
Pull songs for search requests
Manage tags and add songs
Return search results
Music Company Interface
Automatically create search requests
Dataset of Songs

Same Rhythm Radar Context Diagram

1.6 - User Stories

- As a music listener, I want to be able to search songs, so that I get similar songs as a result
- As a music artist, I want to ensure that my songs accurately appear in the output list of general searches; i.e., user search for the top 10 popular songs charting, so that I reach larger audiences.
- As a music streaming service, I want to automatically make search requests, so that I receive recommendations for users
- As a music listener, I want the ability to log in, so that I can have a secure experience.
- As a music listener, I want a friendly interface, so that I can easily lookup new songs
- As a music listener, I want to keep a log of my past searches, so that when I revisit the application, I can reload previous results.
- As a music listener, I want collaborative playlist features, so that I can share my music collection with others

- As a music listener, I want the system to learn from my searches so that I always get personalized recommendations

- As a music listener, I want to be able to quickly make a playlist for an event so that the songs follow a theme.

1.7 - Product Backlog

- Given an LLM is behind the interface of the application, when a music listener inputs a search query, they will then receive results that match up to their request entered by comparing the likeliness of genres, artists, themes, and emotions.

- Given the songs/artists' data will be contained in a dataset, when the data is processed using ML techniques, they will undergo tagging, and then music artists' songs will appear in general searches such as sad songs, rap songs, new songs, and top charting songs.

- Develop functionality to seamlessly integrate SRR into popular music streaming services, ensuring smooth user experience and broad accessibility

- Given the structure of the application, when users first navigate to the application, they will be prompted to either sign up or log in, then they will be granted access to the application.

- Easy to navigate, explore, and interact with recommendation systems, enhancing the overall discovery experience.

- Given the interactive interface, when pre-existing users log in to the application then, their previous searches will be loaded.

- Enable collaborative playlist features, allowing to share and collaborate on playlists with friends and fellow music enthusiasts, and discovery among users.

- Allow the system to keep track of a user's past requests so that future recommendations can be more personalized to the user

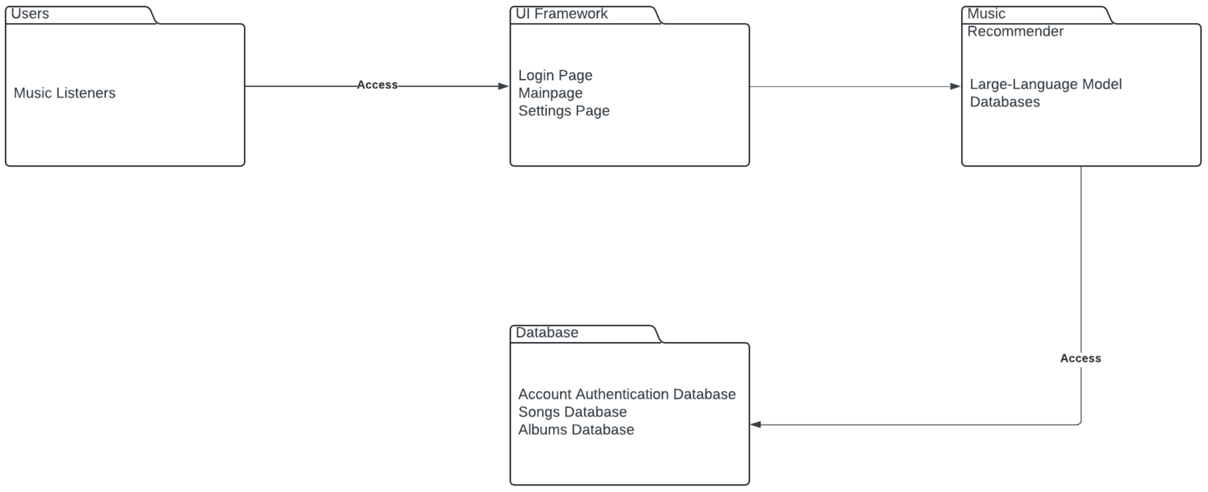
- SSR will look at the dataset and create a playlist consistent with a selected party, event, or theme. It will also ask how many songs are required for the playlist and the duration of the event.

2. Architectural Overview

In the prototyping phase, Same Rhythm Radar engineers bounced around ideas for various architectures and technology stacks before deciding on a server-side architecture that utilizes the Django web framework. In our earlier stages, we discussed three alternative architecture designs: client-server, domain-driven design (DDD), and SOA. We briefly discussed SOA when we thought about how we would modularize the code for reusability efforts, but thinking about the scope of the project, implementing an SOA architectural style would hinder us as our system will not be integrated with any other pre-existing systems. As we researched DDD further, we realized that the architectural style does not fit the problem space of our project.

One of our main implementation efforts for this project is uploading it as a web-based application, which led us to discuss the client-server architectural style. We realized that our program relies more on a server-side architectural style as our users will not be receiving their results from a server-based LLM, but instead, our users' queries will be searched against a database that may be offline at times. Ultimately, this led to our decision for a server-side architectural style as our users' data can be hosted in our relational databases, and our data for our LLM can be stored in its original form while being interacted with by users of our web application.

2.1 Subsystem Architecture



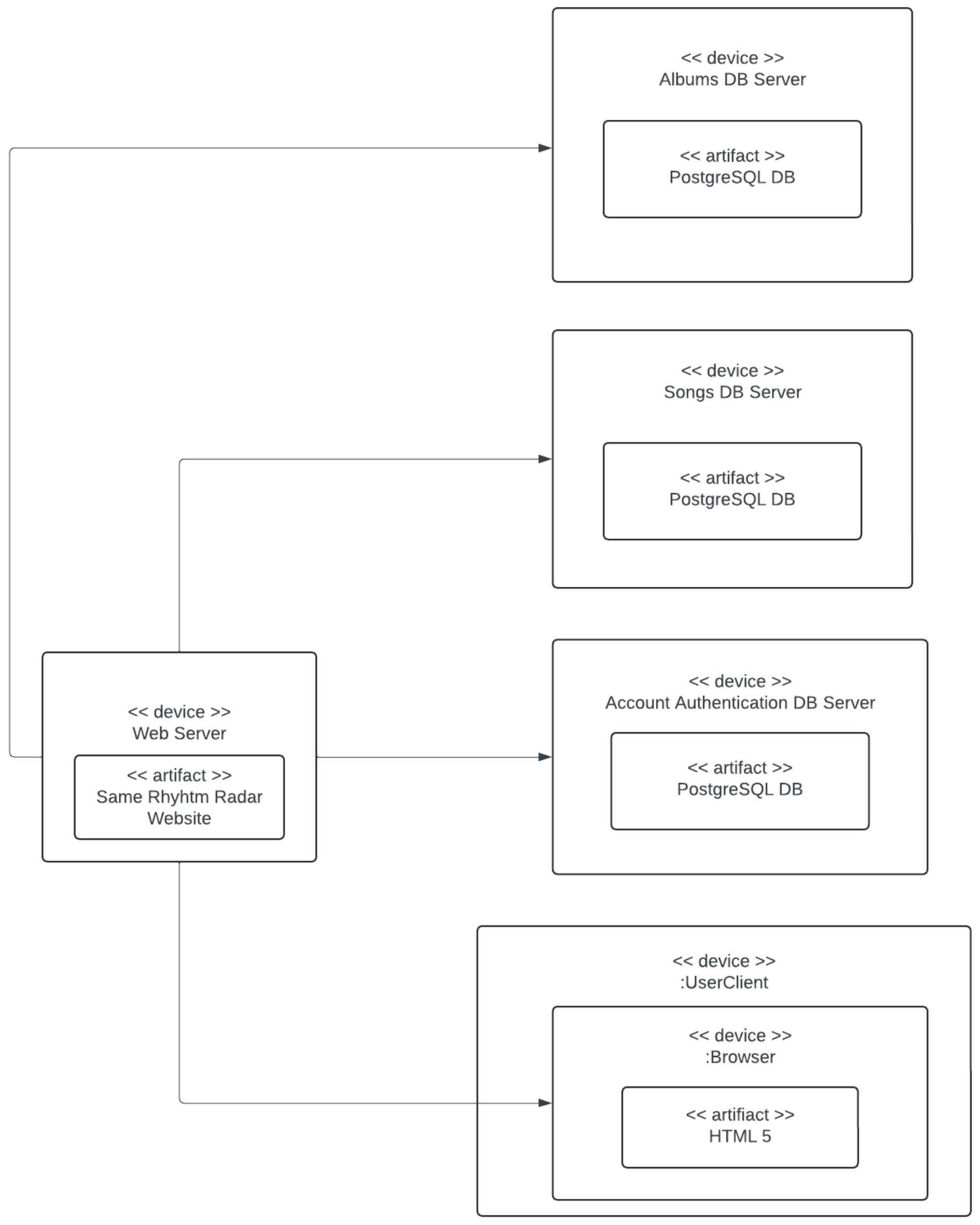
UML Dependency Package Diagram: High-Level Overview of Same Rhythm Radar

The above diagram offers a look at how each subsystem interacts with one another based on the package that best represents their use case. At our system's highest level, the external environment that interacts with our system is users, mainly music listeners seeking recommendations. We've discussed in earlier documents that our users involve music listeners, artists, and music streaming companies, and it does; however, in a context-level situation, music listeners are our system's most important relationship.

Three main databases communicate with our system to process real-time requests: the account authentication database, the songs database, and the albums database. With where we are currently in our development phase, our developers found that PostgreSQL proves to be the strongest choice as far as processing and handling our data; it handles scalability, wide-use functionality, and collaboration with Python, JavaScript, HTML/CSS which are libraries used to support our system. Our main components are the LLM, two relational databases, and the web pages that allow for site visitors.

In a more in-depth code level, our code emphasizes modularity to enhance code readability, and time complexity and promote the design rules for software development. Our main system will be comprised of different code files that will communicate with one another; for example, our web pages have been built using separate HTML files that communicate with their own CSS files. As users input data on the HTML web pages, the input is saved and stored in Python variables to either be saved in the account authentication database or used to make a connection with our music recommendation LLM. Our Python code will be comprised of different files that serve different purposes: the build-out of our large-language model, communication with the HTML files and user input, and communication with our relational databases.

2.2 Deployment Architecture



UML Deployment Diagram: Same Rythm Radar

The above diagram displays how our web application interacts with the other features of our system. As mentioned in the above section, Same Rythm Radar is being built using a server-side architectural style. To build out and later deploy our application, we will shell our application within the Django web framework. Django offers a seamless deployment as it supports two web interfaces, WSGI and ASGI. WSGI is the main Python standard for communicating between web servers and applications, but it only supports synchronous code. ASGI is a newer, asynchronous standard that will allow our Django website to use asynchronous Python features, and asynchronous Django features as they are developed.

Django allows us to ensure Same Rythm Radar remains scalable, robust, secure, maintainable, and portable which is most important as this feature will allow our application to be run on many platforms, not just Windows, Linux, or MacOS. After our application has been deployed, communication between the two relational databases will remain intact and in fact, be seamless as the Django library shares a great integration relationship with PostgreSQL.

2.3 Data Model

For Same Rhythm Radar, we have opted to employ PostgreSQL as the backend database management system for storing data persistently. Our database schema comprises three principal tables: 'Users', 'Songs', and 'Albums'. The 'Users' table includes fields for username and password, facilitating user authentication. Meanwhile, the 'Albums' table encompasses attributes such as album_name, artist, genre, release_date, and number of listens, providing comprehensive information about albums. Lastly, the 'Songs' table incorporates details like title, artist, genre, release_date, and number of listens, along with a foreign key relationship to the 'Albums' table, establishing a many-to-one relationship between songs and albums. This structured approach to data storage enables efficient organization and retrieval of information, ensuring the seamless functioning of our web application.

2.4 Global Control Flow

Procedural or event-driven: The system primarily follows an event-driven model of control flow. It waits for user-generated queries and responds by populating a container with relevant song recommendations. Users can generate queries in any order, and the system dynamically updates the response container based on the received queries. For example, a user might input a query for "uplifting songs," and the system responds with a list of uplifting songs without the need for sequential steps.

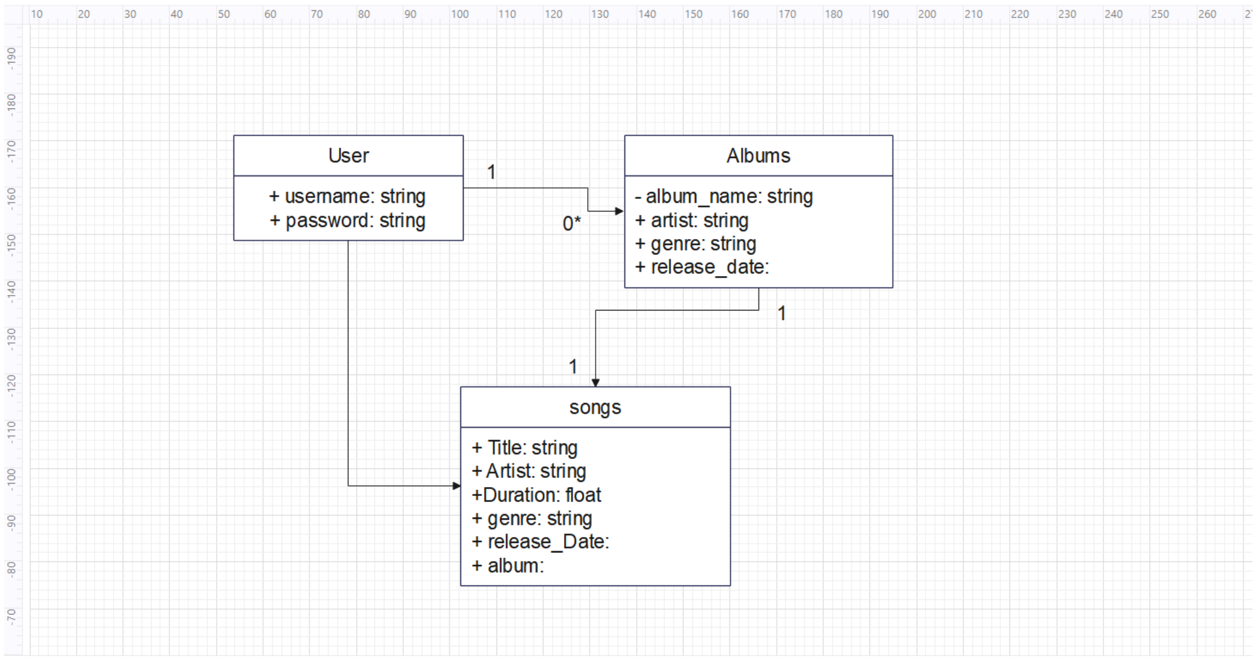
Time dependency: The system's control flow is not time-dependent. It responds to user queries without real-time constraints. Each query triggers an event, and the system generates responses based on these events. There are no periodic actions or time constraints associated with the system's operation.

Concurrency: While the system does not require multiple threads for concurrent execution, it may utilize asynchronous processing to enhance performance. For instance, when handling multiple user queries simultaneously, the system may employ asynchronous methods to fetch song recommendations efficiently. However, the core functionality remains single-threaded, with concurrency optimizations implemented at the application level rather than through multiple threads.

3 Detailed System Design

As explained in Section 2, Architectural Overview, one of the key components of our web application is user interaction with the system. Music listeners will generate most of the traffic that our web application will experience, which is why we decided to focus primarily on their experience in the detailed system design section of our document. Our main goal, as shown in our user stories and product backlog (Sections 1.6 and 1.7), is to provide our users with an easy-to-navigate user interface and web experience, so our detailed system design discusses the relationship between the user database and our albums and songs database and the flow experience for a user as they navigate the web application.

3.1 Static view

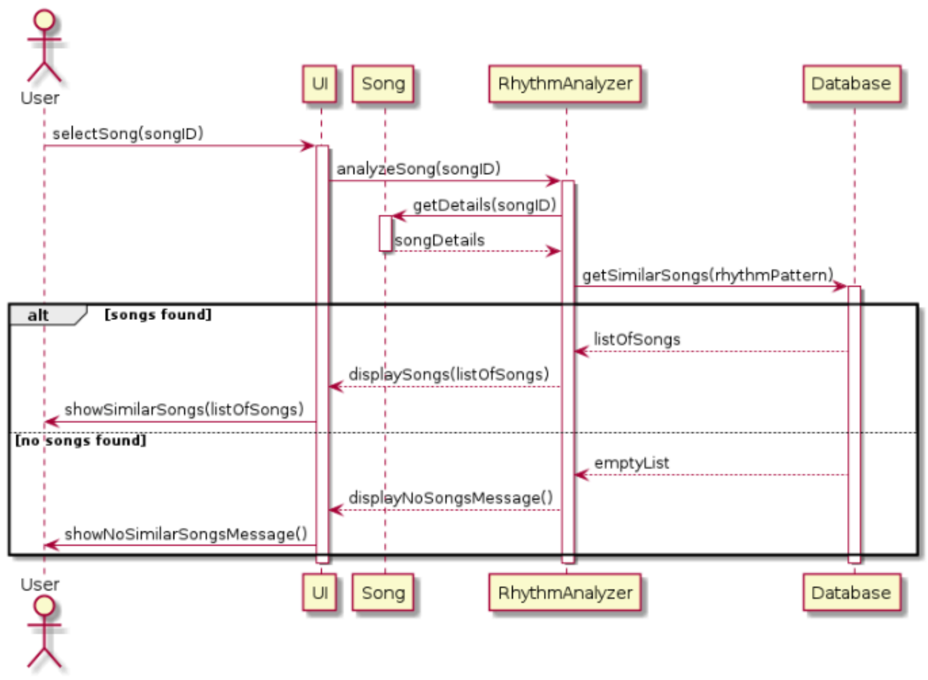


UML Class Diagram: Same Rhythm Radar

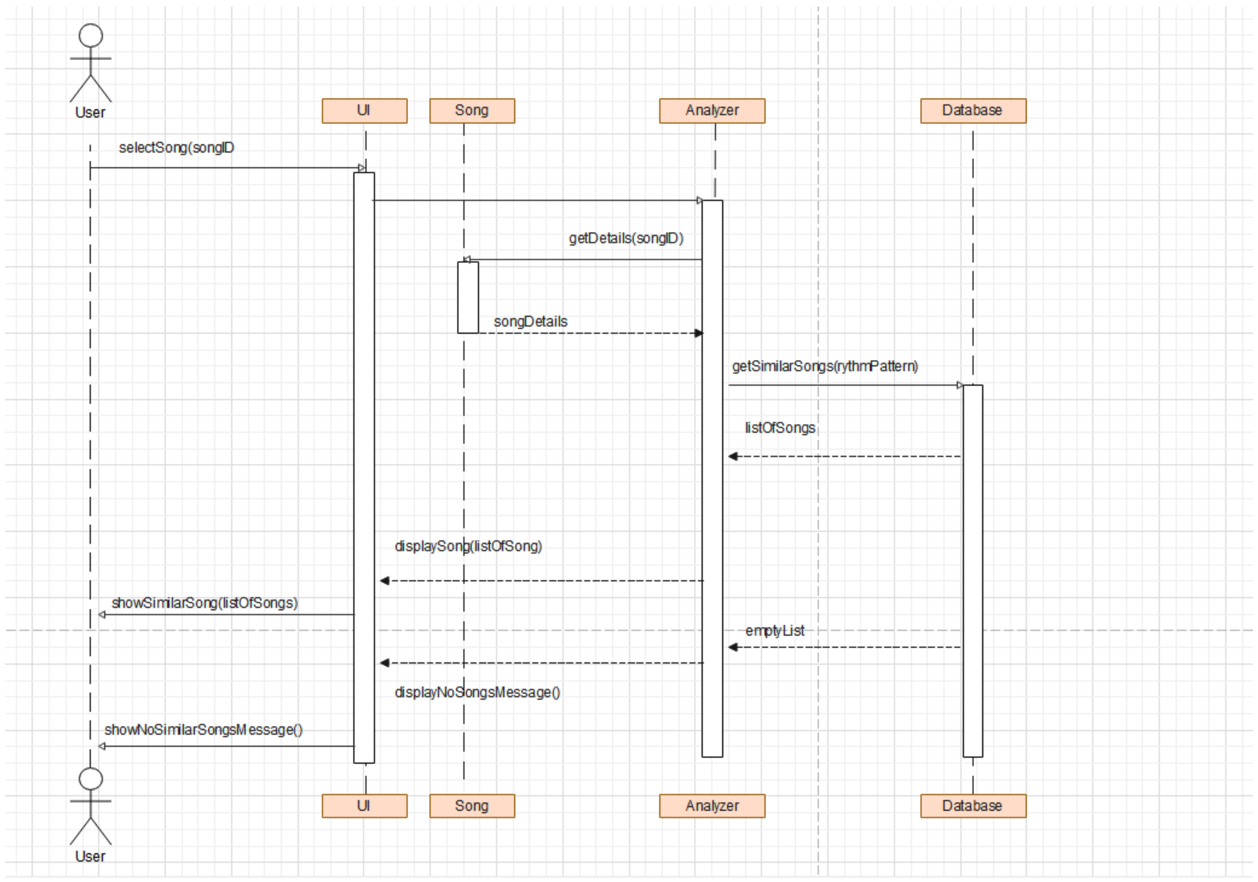
The above diagram shows the main classes between our user (interface) and their entered queries against our relational database that hosts our album and song data. As discussed in section 2.3 Data Model, two tables will comprise our album information and song information. As the user enters a query request, it is searched against our database tables, and since the tables will follow an organized many-to-one key relationship, data retrieval will be seamless, and the user will expect an outputted response from our system.

3.2 Dynamic view

the User, UI, Song, RhythmAnalyzer, and Database based on your class diagram.



UML Sequence Diagram 1: Same Rhythm Radar



UML Sequence Diagram 2: Same Rhythm Radar