SpUStify

Software Architecture Document

Version 1.1

Revision History

| **Date** | **Version** | **Description** | **Author** |
| --- | --- | --- | --- |
| 10/7/2023 | 1.0 | Initial version | Dương Trường Bình |
| 05/08/2023 | 1.1 | Revised version in PA4 | Dương Trường Bình |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[**1. Introduction 4**](#_heading=h.gjdgxs)

[**2. Architectural Goals and Constraints 4**](#_heading=h.30j0zll)

[**3. UseCase Model 5**](#_heading=h.1fob9te)

[**4. Logical View 6**](#_heading=h.2et92p0)

[4.1 Component: Presentation 8](#_heading=h.tyjcwt)

[4.2 Component: Controller 10](#_heading=h.9t78hgk7fky6)

[4.3 Component: Data access 13](#_heading=h.y57vlvcwahy2)

[4.4 Component: Model 15](#_heading=h.hk4saelsnl5p)

[4.5 Component: Serializer 16](#_heading=h.l64om09ls9rk)

[**5. Deployment 18**](#_heading=h.3dy6vkm)

[**6. Implementation View 20**](#_heading=h.1t3h5sf)

Software Architecture Document

# Introduction

The introduction of the **Software Architecture Document** provides an overview of the entire **Software Architecture Document**. It includes the purpose, scope, definitions, acronyms, abbreviations, references, and overview of the **Software Architecture Document***.*

# Architectural Goals and Constraints

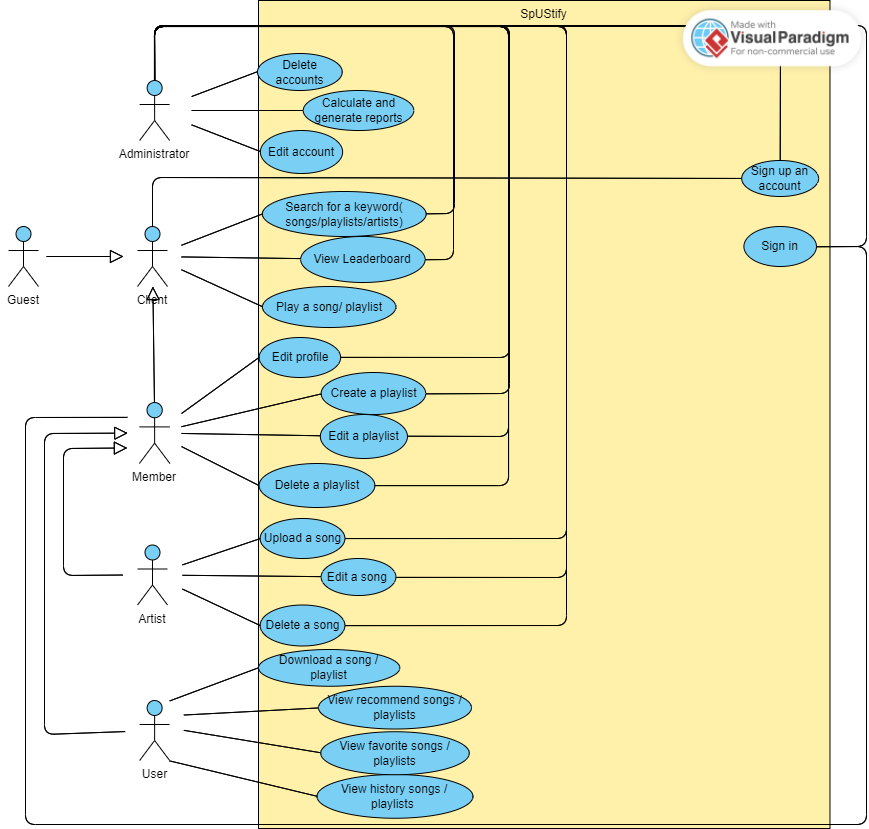
Goals:

* Scalability: Design for handling a large number of concurrent users and music streams without performance degradation.
* Performance: Ensure fast loading times and responsive user interactions for an optimal user experience.
* Security: Implement robust data protection and authentication mechanisms to safeguard user information.
* Privacy: Treat user data with confidentiality and comply with data protection regulations.
* Modularity and Reusability: Create a modular architecture to facilitate easy maintenance and extension of functionality.
* Maintainability: Prioritize clean code practices, thorough documentation, and well-defined interfaces for efficient maintenance.
* Integration with ThirdParty Services: Allow smooth integration with external services for enhanced features and functionality.
* Realtime Updates: Provide real-time updates for dynamic features and user notifications.

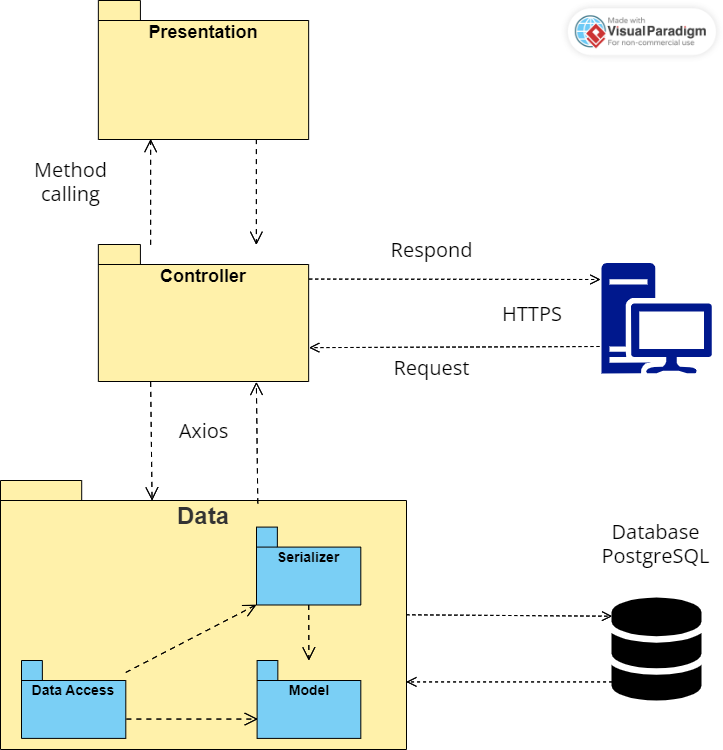
Constraints:

* Backend Technology: Use Python (Django) and SQL for the backend development.
* Frontend Technology: Utilize HTML, CSS (Tailwind CSS), and JavaScript (ReactJS) for the front end.
* Development Team Structure: Organize a team of backend and frontend developers, UI/UX designers, and quality assurance engineers.
* Deployment Schedule: Follow a predefined schedule for timely website delivery with a balance between functionality and quality.
* Legacy Code Considerations: Refactor and document any integrated legacy code to align with the overall architecture.
* Cost Constraints: Consider cost-effectiveness in infrastructure, hosting, and third-party services selection.

# UseCase Model



# Logical View

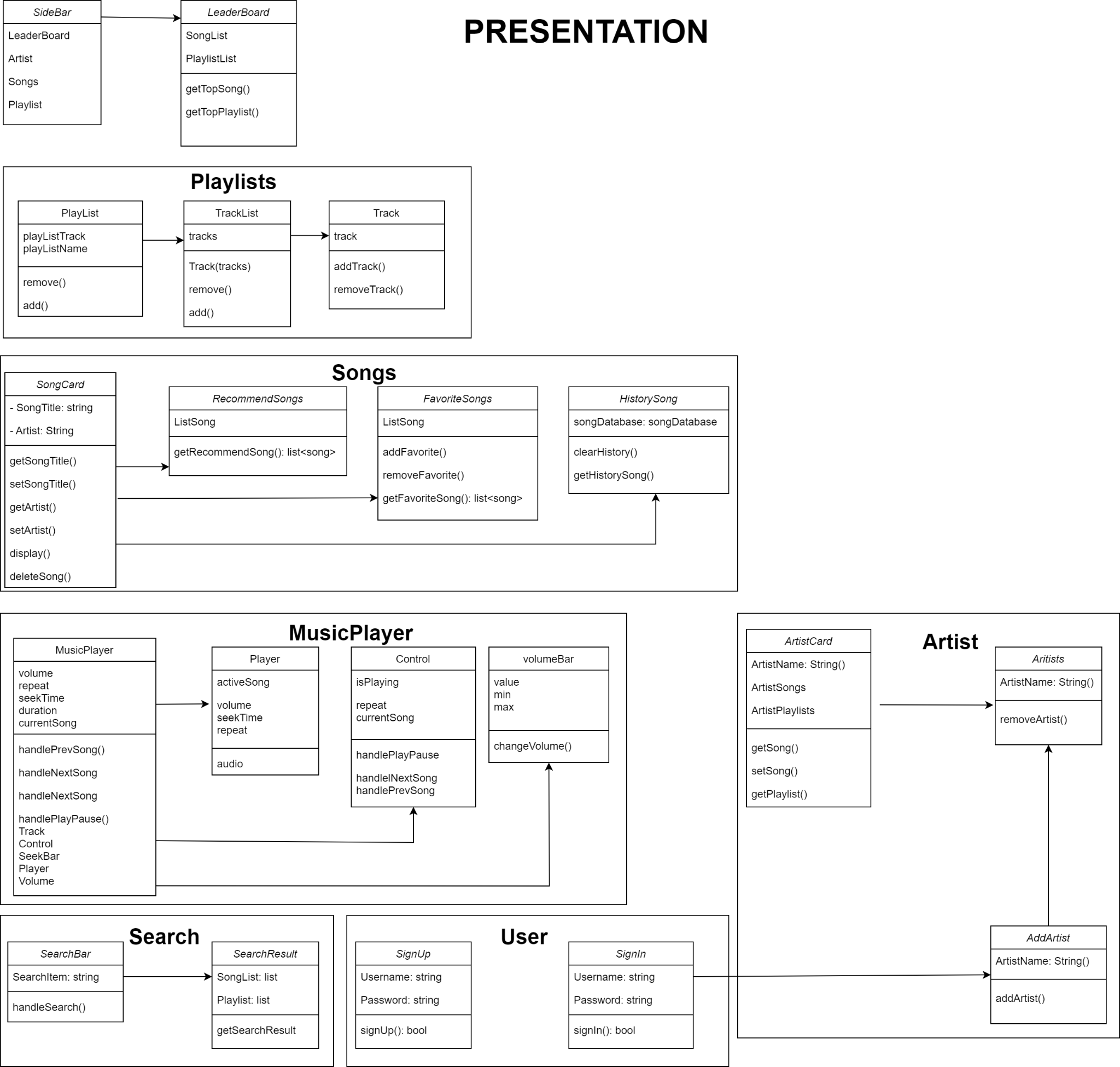


The website's architecture is composed of the following essential packages:

* **Presentation:** Component Presentation in our application refers to the user interface and visual elements of the music web application that users interact with, developed with ReactJS and Tailwind CSS. It involves designing and presenting the various components of the application such as Song, SideBar, LeaderBoard, etc. Each of these components is responsible for displaying specific information and providing functionalities to enhance the user experience.
* **Controller:** The component Controller in our application refers to the part of the application responsible for handling the logic, data manipulation, and connecting the presentation component and the backend services or data sources. It ensures that the UI renders the data expected from the database when users interact with the application's interface.
* **Model:** plays a crucial role as it defines the data models used to represent various entities in the application. By creating classes that inherit from Django's models. Model, developers can articulate the structure of database tables and organize the data efficiently. Each class in models.py corresponds to a specific data entity, such as users, artists, songs, or playlists, and the attributes within these classes represent the fields of the corresponding database tables. This enables developers to interact with the database through Python code, performing essential operations like creating, reading, updating, and deleting data (CRUD operations).
* **Serializer:** the focus is on transforming complex data, such as Django model instances, into formats that are easily exchangeable, such as JSON. Serializer classes within this file handle data serialization, validation, and deserialization. By converting data into JSON format, the application can seamlessly communicate with the frontend and backend, facilitating smooth data exchange between the two layers. Moreover, serializers play a significant role in ensuring that data sent from the client side is transformed into valid Python objects that can be efficiently processed and stored in the database.
* **Data Access**: is the bridge between the front and backend, handling user requests and generating suitable responses. Views can be implemented as functions or classes and determine the content displayed to users when they access specific URLs (endpoints). By processing user input and interacting with models and serializers, views orchestrate the application's logic and provide the appropriate data to be rendered on the front end. They are instrumental in delivering dynamic content, fetching data from the database, and returning JSON responses or rendering HTML templates to users, ultimately defining what the users see and interact with on the application.

## 

## Component: Presentation



1. **Songs:** The Song group is responsible for displaying detailed information about songs, suggesting suitable songs, managing the list of favorite songs, and keeping track of search history

* SongCard:
  + Function: Show the information about songs like song title, artist, etc.
  + Flow: The information of the song will be display at SongCard, when users interact with the Song Card, for example, by clicking the heart button, this event is forwarded to the Favorite song component to add the song to the user's list of favorites
* RecommendSongs:
  + Function: The recommended songs will be displayed in the Recommend Song section for users to explore and play music
  + Flow: Using information about download counts, favorite counts, and play counts to recommend songs that match the user's preferences and tastes. Based on this information and specific algorithms, the recommended songs will be displayed in the Recommend Song section for users to explore and play music
* FavoriteSongs:
  + Function: Managing the list of songs that users have marked as favorites. This helps users easily search for and access their favorite songs
  + Flow: Favorite Songs receives information about favorite songs when users perform actions to add or remove songs from the favorites list
* HistorySongs:
  + Function: Tracking the user's music listening history, including a list of the most recently played songs
  + Flow: History Song receives information about the songs that have been played from the Song Card. It stores a list of the user's recently listened to songs.

1. **Artists:** provides information and content related to a specific musician, band, or musical group. The Artist class is essentially a userfacing interface or view where users can access details about their favorite artists and explore their music.

* ArtistCard: it is a layout to display all information of an artist like name, ID, songs, their playlists, their followers. Users can easily identify and learn more about the artist they are interested in.
* AddArtist: it is a form for Users who are singers or create songs can sign up as artists which can upload their songs on our application.

1. **Playlists:** it is a curated collection of songs or audio tracks grouped together based on a specific theme, mood, genre, or personal preference. Playlists are an essential feature that allows users to organize their favorite music, discover new songs, and create personalized listening experiences.

* Tracklist: The tracklist represents the ordered list of songs or audio tracks within a playlist or album. Users appreciate a clear and organized tracklist that allows them to see the sequence of songs.
* Track**:** it is a layout to display all information of a song like name, ID, lyrics, and audio. Users can easily connect to a particular song detail in the current playlist.

1. **Music Player:** This class allows users to play and control audio tracks, such as songs or podcasts. When integrating a music player into an application, we can implement various functionalities to create a rich and interactive music listening experience for users.

* Player: offers standard playback controls, including play, pause, stop, and seek (allowing users to move to a specific time in the track).
* Control: offer standard playback controls, including skip forward, and skip backward.
* Volume Bar: Users can be able to adjust the volume of the audio playback using volume controls.

1. **SideBar:** A navigation bar on the web for clients to use different web features.

* Function: The sidebar contains 4 features: Leaderboard, artist, songs, and playlist. When the client chooses one of those features, the corresponding page will show up.
* Flow: When the client chooses one of those features, the corresponding page will show up.

1. **LeaderBoard:** Display the top-liked songs and artists.

* Function: Display the top liked songs and artists to the clients.
* Flow: When the client clicks the ‘Leaderboard’ word in the sidebar, display the leaderboard.

1. **Search:** The Search group is responsible for searching the songs and the playlists and then displaying them to the client.

* SeachBar:
  + Function: Get the search word from the client.
  + Flow: The client inputs the search word then the web will display the search result page.
* SearchResult:
  + Function: Display the search results based on the search word in the search bar.
  + Flow: Receive the search word from the search bar then get information from the database and display it.

1. **User:** The User group is responsible for the client’s sign-in and signup.

* SignUp
  + Function: Display a form for clients to input information for their new account.
  + Flow: The clients input information and then inform the client about their registration.
* SignIn
  + Function: Display a form to get a username and password of users.
  + Flow: The client's input information then inform the client about their sign-in.

## Component: Controller

1. **Songs:** The Songs controller in the music web application is responsible for managing various song-related operations.

* *UploadSong*: This class handles the process of uploading new songs to the application. It receives song files and metadata from the front end and stores the data in the backend database.
* *DeleteSong*: This class manages the functionality to remove a song from the application. It receives the song ID and name, then deletes the corresponding song data from the database.
* *getRecommendSong*: This class retrieves a personalized list of recommended songs for the user based on their listening, likes, and download history. It aims to enhance the user experience with tailored song suggestions.
* *getFavoriteSong*: This class retrieves the list of songs marked as favorites by the user. It allows the front end to display the user's favorite songs for easy access.
* *removeFavoriteSong*: This class handles the functionality to remove a song from the user's list of favorite songs. It receives the user ID and song ID from the front and removes the corresponding song from the list of favorites.
* *getHistorySong*: This class retrieves the user's listening history, including a list of recently played songs.
* *clearHistorySong*: This class manages the functionality to clear the user's listening history. It allows the user to reset their listening history and start fresh.

1. **Artists:** it typically refers to the individual musicians, bands, or musical groups who create and perform the music. We work with artists in such an application involving handling and presenting information related to these musical creators.

* *getArtistinfo*: Developer can access artist-related data from APIs or databases, including details such as the artist's name, biography, images, genres, and other relevant metadata.
* *removeArtist*: responsible for removing an artist from a user's preferences, playlists, or any other associations within the application. This would involve updating the database, removing the artist from the user's followed artists list, or removing the artist's tracks from the user's playlists.
* *UpdateUserToArtist*: We would convert users to artists by moving their profile information to the artist database, which would grant them access to additional features in our application.

1. **Playlists**: allows us to interact with and manage playlists programmatically. It provides a set of functions and methods that enable you to perform various operations related to playlists.

* *removeTrack*: we fetch the playlist from the database, remove the specified track from the playlist's tracklist, and update the playlist object in the database.
* *AddTrack*: We fetch the playlist from the database, add the new track to the playlist's tracklist, and update the playlist object in the database.
* DeletePlaylist: we find and remove the playlist object from the database, ensuring it is no longer associated with the user.
* *CreatePlaylist*: it takes input parameters such as the user's identifier (ID), the playlist name, description, and an array of initial track identifiers (IDs) if the user wants to add tracks while creating the playlist. Then, we implement the logic to create a new playlist object in the application's database, associate it with the user, and optionally add the specified tracks to the playlist.
* *SavePlaylist*: we implement the logic to update the playlist object in the database with the changes made by the user, such as adding or removing tracks and updating playlist metadata.

1. **LeaderBoard:** the leaderboard controller is responsible for showing the top songs and artists to the client.

* getTopSong: This class handles the process of getting top song information and displaying them to the client.
* getTopPlaylist: This class handles the process of getting top playlists information and displaying them to the client.

1. **User:** handles the sign-in/sign-up of the client. It allows clients to become a user who will have more access to user-only features.

* createUser: This class will get the information from the user and then check in the database if the information is valid. If it is valid, create a new user account with the information in the database
* authenticateUser: this class will get the username and password from the user and then check in the database if the information is valid.

1. **SearchEngine:** handles the search feature of the web, and returns the result based on the search keys.

## Component: Data access

Importing essential modules and packages is crucial for constructing API views. These components include serializers to convert complex data types into JSON, authentication classes for managing user login and registration processes, and models representing the application's database structure.

The **RegisterAPI** permits new users to create accounts and register, while the **LoginAPI** facilitates user login through token-based authentication provided by Django Knox.

The HomeViewAPI and HomeFeaturesAPI serve as versatile API views central to the application's homepage features, offering a diverse set of functionalities:

* Users can search for songs and playlists based on their queries.
* The API displays a leaderboard showcasing song likes and listens.
* Users can retrieve their preferred and previously played songs and playlists.
* Personalized song and playlist recommendations enhance user interaction and content suggestions.

For administrative purposes, the **AccountsViewAPI** allows listing all user accounts, and the **EditAccountAPI** permits the deletion of specific accounts, simplifying user account management.

The **ProfileViewAPI**, **ArtistsViewAPI**, **SongsViewAPI**, and **PlaylistsViewAPI** classes are focused on information-related functionalities, including searching based on user queries and retrieving detailed information.

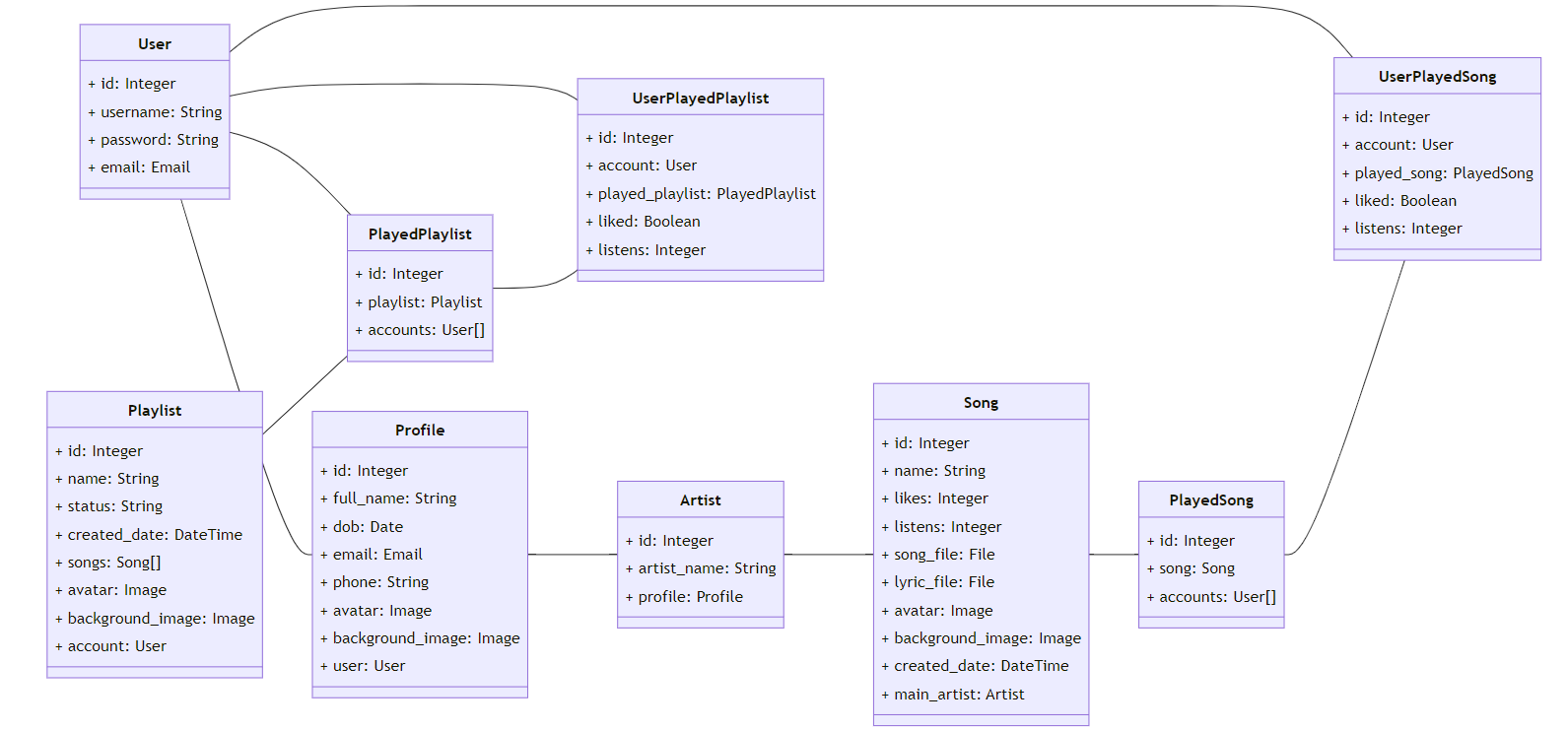
The **CreateProfileAPI**, **CreateArtistAPI**, **CreateSongAPI**, and **CreatePlaylistAPI** classes are designed for creating new information entries.

The **EditProfileAPI**, **EditArtistAPI**, **EditSongAPI**, and **EditPlaylistAPI** classes handle information-related functionalities such as updating or deleting existing data. Note that the ProfilePageAPI only updates information and does not delete it.

Lastly, the **AddSongToPlaylistAPI**, **PlaySongAPI**, **LikeSongAPI**, **PlayPlaylistAPI**, and **LikePlaylistAPI** classes enable users to play music, add it to their favorites, and maintain a history of played music by using features like liking songs and tracking play history.

## 

## Component: Model



1. **User Model:**

* Key model builtin with Django.
* Represents users in the system.
* Manages authentication and access control.

1. **Profile Model:**

* Created to complement user information.
* Includes a profile picture and biography.
* Establishes a one-to-one relationship with the User model.

1. **Artist Model:**

* Represents artists in the music system.
* Stores essential information about artists (names, biography).

1. **Song Model:**

* Holds details about songs in the application.
* Includes attributes like title and duration.
* Linked to the Artist model via ForeignKey and ManyToManyField.

1. **Playlist Model:**

* Represents playlists within the application.
* Contains information like title and description.
* Has a relationship with the User model via ForeignKey (determines the owner of the playlist).
* Utilizes ManyToManyField to manage the songs within each playlist.

1. **PlayedSong and PlayedPlaylist Models:**

* Store information about songs and playlists played by users.
* Provide insights into user interactions with music content.

1. **UserPlayedSong and UserPlayedPlaylist Models:**

* Determine whether users liked the played songs and playlists.
* Capture user preferences and interactions with specific music content.

## Component: Serializer

1. **RegisterSerializer:**

* Responsible for processing user registration data.
* Takes user details (username, email, password) as input and securely stores the password.
* Ensures that sensitive information like the password is written only and not visible in the serialized output.
* Enables users to create new accounts securely and efficiently.

1. **SongSerializer:**

* Converts Song objects into JSON format.
* Includes fields like names.
* Utilizes a SerializerMethodField (main\_artist) to retrieve and display associated artists for each song.
* Presents comprehensive information about songs, including details about contributing artists, facilitating client access to song-related data.

1. **PlaylistSerializer:**

* Handles the serialization of Playlist data models.
* Captures attributes such as name and status (public or private) for better readability.
* Utilizes a SerializerMethodField (songs) to retrieve the list of songs within each playlist.
* Enhances the user experience by providing detailed information about playlists and their contents.

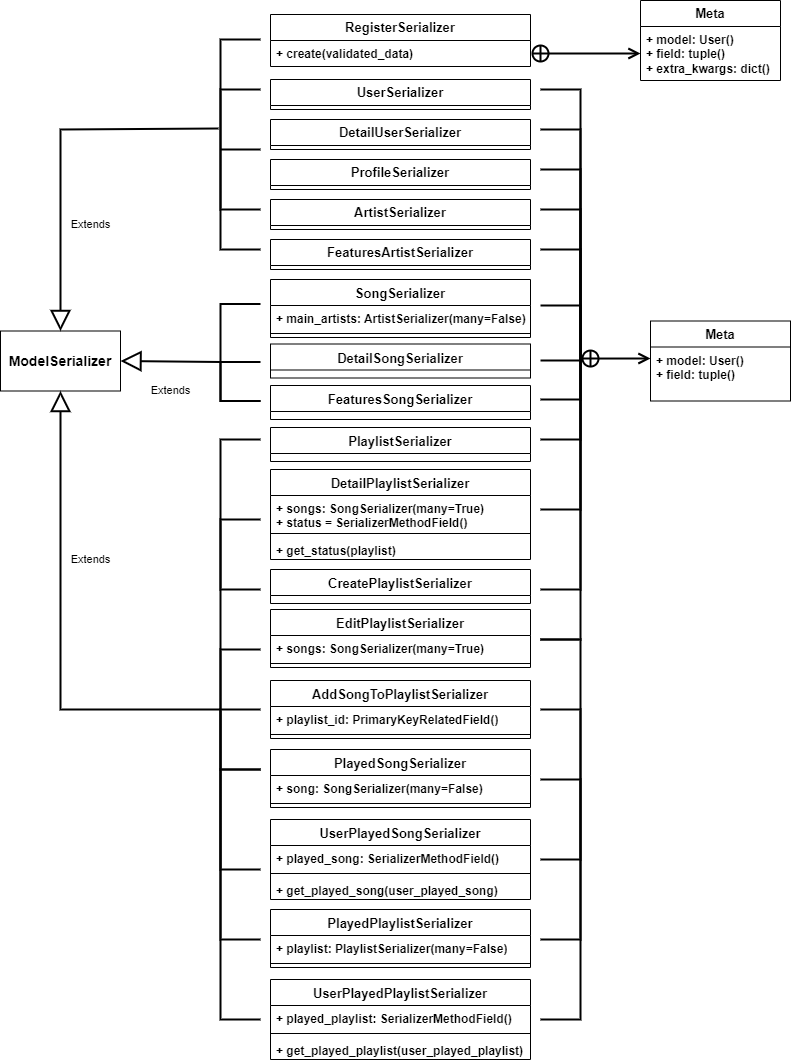
The mentioned Serializer classes play a significant role in generating data for essential web functions, such as the leaderboard, listening history, favorite history, and listening recommendations.

1. **PlayedSongSerializer and UserPlayedSongSerializer:**

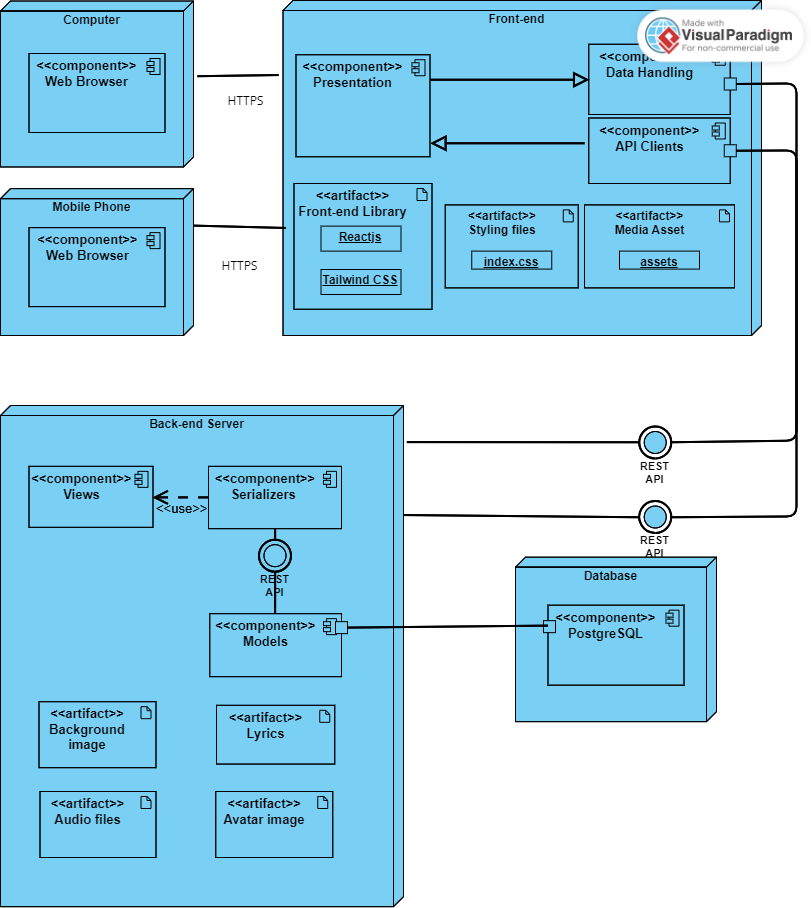
* Handle data serialization for songs that users have interacted with.
* Provide a comprehensive view of the songs users have played, including details about the contributing artists.
* Simplify the serialization process and ensure consistent data representation.
* Enhance the user experience and facilitate seamless data exchange.

1. **PlayedPlaylistSerializer and UserPlayedPlaylistSerializer:**

* Handle data serialization for playlists that users have interacted with.
* Retrieve information about playlist names and the songs within each playlist.
* Effectively present playlists along with their associated songs, offering a comprehensive view of user interactions with playlists.



# Deployment



The deployment diagram illustrates the distribution of components and artifacts across various nodes in the system. In this system, we have main nodes:

1. **Computer & Mobile phone**:

These nodes represent the devices (computers and mobile phones) on which the web application can be accessed. They have the Web Browser component, allowing users to interact with the application through a web browser.

1. **Front-end Node**:

This node includes 3 components: Presentation, API Clients, and Data Handling.

* Presentation: responsible for displaying the web application's user interface. It interacts with the API Clients component to receive data and updates from the backend.
* API Clients: gets input from the client through their interactions on the web application. Then it sends data to Back-end. For example User information, songs that Artist uploads, etc…
* Data Handling: It receives data from the backend and sends the processed results to the presentation to display UI.

This node also utilizes some artifacts which are:

* Front-end library:
  + ReactJS, which is a JavaScript front-end library, is used to construct the web application
  + TailwindCSS, which is a CSS library, is used to style the components in the web application for better display.
* Styling files: include an ‘index.css’ file for styling components in the web application.
* Media Asset: include an ‘assets’ file which holds all the images, icons, videos,... to display on the web application.

1. **Backend Node**: The Backend node handles the server-side operations and processing of data.

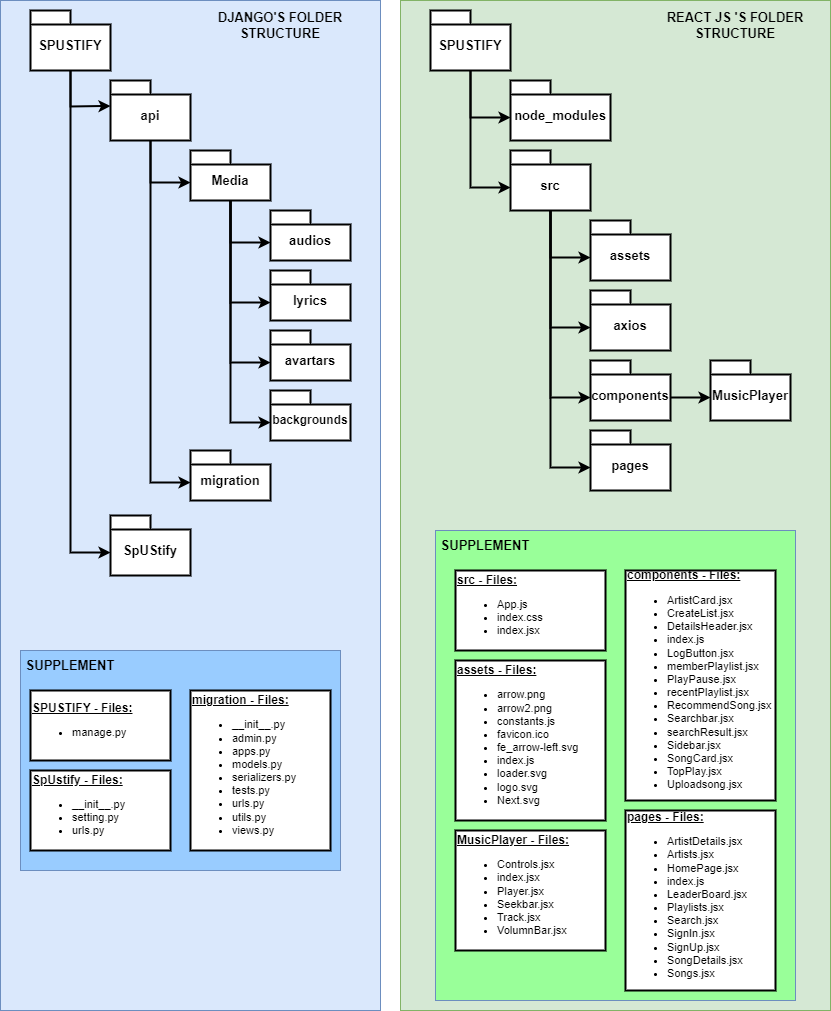
It consists of several components:

* Serializers: responsible for converting complex data types, such as models, into a format that can be quickly rendered by the front end. They are used to communicate with the Views component through REST API.
* Models: defines the data structures and handles the interaction with the Database node through the PostgreSQL component..
* Views: handles the interaction between the front and the backend, processing user requests and sending appropriate responses.
* Artifacts used by the Backend Node:
  + Audio Files, Lyrics, Background Image, and Avatar Image: These are likely resources used in the application that are stored and managed by the backend.

1. **Database Node with PostgreSQL Component**:

The Database node represents the PostgreSQL database used to store and manage the application's data. It interacts with the backend to handle data storage and retrieval.

# Implementation View

**