Group02

Beatify
Software Architecture Document

Version 1.1

Revision History

Date	Version	Description	Author
28/11/2024	1.0	This document outlines the software architecture, covering the system's structure, components, deployment, and implementation. It includes architectural goals, use-case models, logical views of key components, and deployment details. The aim is to provide a clear framework for development, ensuring scalability, maintainability, and performance.	- Lê Gia Huy: Architectural Goals and Constraints, Logic View: Architecture Diagram. - Võ Ngọc Khoa: Architectural Goals and Constraints, Component: Controller. - Phạm Hà Hiếu: Introduction, Component: View. - Trần Trung Hiếu: Introduction, Component: Route. - Nguyễn Duy Bảo: Introduction, Component: Model.
13/12/2024	1.1	Fix class diagram for Component: Controller.	Võ Ngọc Khoa
13/12/2024	1.1	Add Deployment and Implementation View	Lê Gia Huy

Table of Contents

Software Architecture Document

1.Introduction

1.1 Purpose

This article offers general information about the system's operation and functionality as a whole. In addition, it explains how the system architecture's use cases and functionalities work.

1.2 Scope

➔ Give complete instructions on how the system runs and ensure that all functional and non-functional requirements are met.

➔ Such a document is used in the "Beatify" application development process.

1.3 Definition

None

1.4 Acronyms

None

1.5 Abbreviations

None

1.6 References

➔ Vision document.

➔ Use-case specification document.

1.7 Overview

➔ Architecture Goals and Constraints: *describes the software requirements and objectives that have some significant impact on the architecture.*

➔ Use-case Model: *the use case diagrams that are already modeled and presented in the use-case specification document.*

➔ Logical View: *the architecture with components and relationships among them*.

➔ Deployment: *how the system is deployed by mapping the components.*

➔ Implementation View: *folder structures for our code for all components.*

2.Architectural Goals and Constraints

Architectural Constraints:

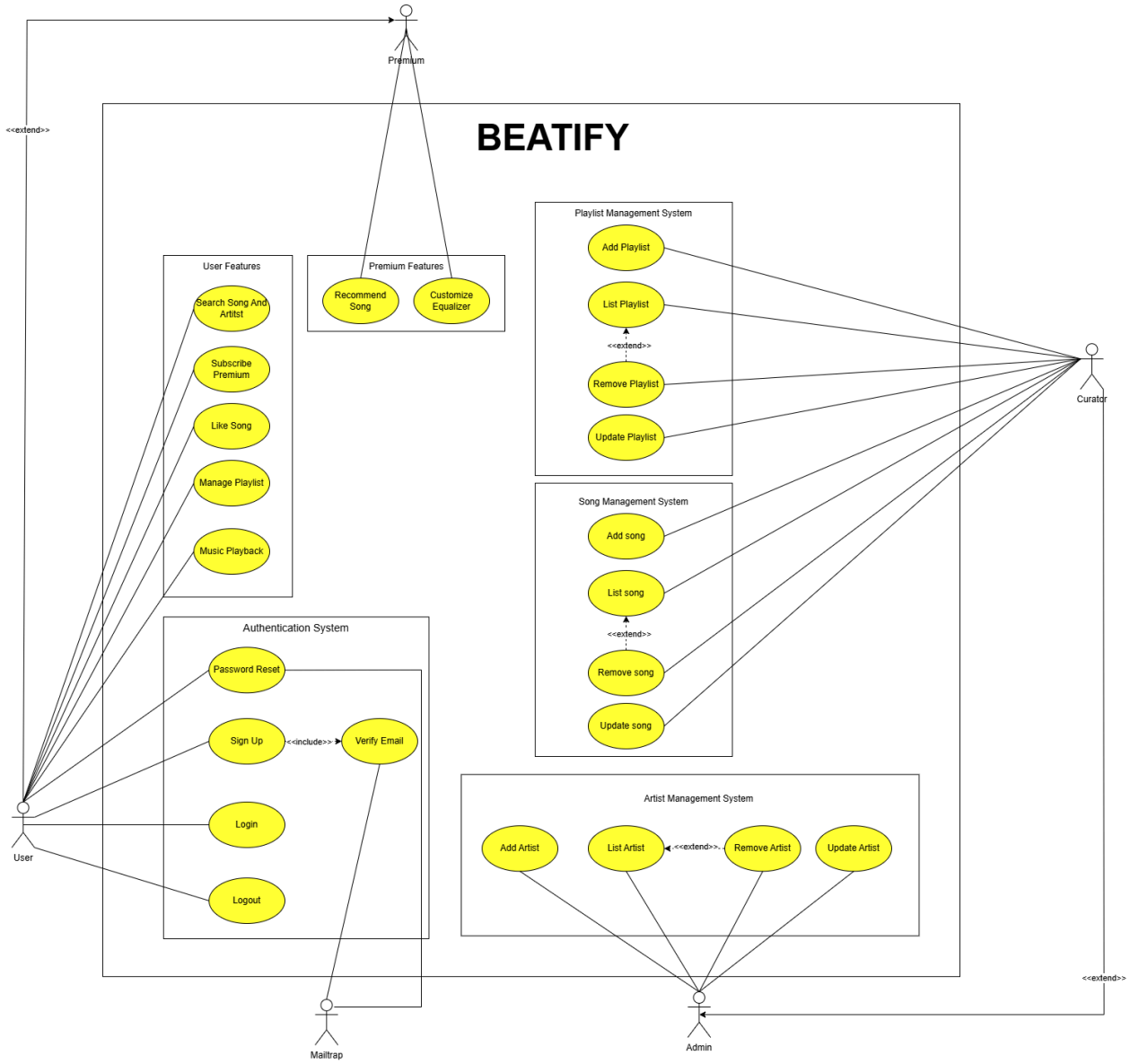
- **Programming language and framework:**
  - Frontend: NextJS, React, Tailwind CSS
  - Backend: NextJS, Express
- **Database:** Utilizes MongoDB to manage user information, songs, albums, playlists, and other related data.
- **Payment Processing:**Integrated with Stripe to handle payment transactions and manage subscription plans (free and premium).
- **User Interface Design:** Designs must be simple and clear to ensure a comfortable user experience.
- **Application environment:** Web.
- **Programming environment:** Visual Studio Code.

Additional Constraints:

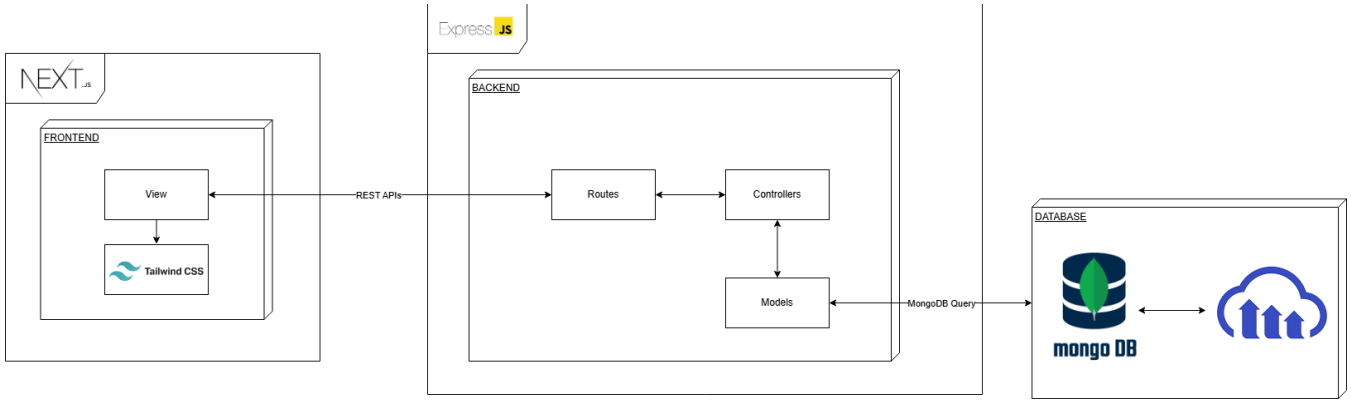
- **Web Standards Compliance**
  - Adhere to the latest W3C web standards to ensure compatibility, accessibility, and cross-browser support.
- **Platform Requirements**
  - The application will be primarily web-based, optimized for deployment on a reliable cloud platform such as Vercel.

    - Ensure compatibility with modern browsers, specifically Google Chrome, Mozilla Firefox, Safari, and Microsoft Edge (latest two versions).
- **Performance Objectives**
    - Achieve a response time of under 2 seconds for the main interface under standard broadband conditions.
    - Support up to 100 concurrent users without significant performance degradation.
- **Security Constraints**
    - Implement robust measures to protect user data and ensure secure financial transactions, meeting industry standards and legal compliance requirements.
- **Scalability and Availability**
    - Design for scalability to accommodate future growth in user base and data volume.
    - Maintain 99.9% uptime to ensure consistent availability for users.
- **Usability and Accessibility**
    - Prioritize an intuitive and user-friendly interface, enabling seamless navigation and interaction for users.
    - Incorporate accessibility features to ensure inclusivity for users with diverse needs.
- **Reliability Requirements**
    - Build mechanisms for error handling, ensuring data consistency and preventing loss during system failures.
- **Dependencies and External Services**
    - Stripe integration for payment processing, requiring compliance with their API standards and terms of service.
    - AI services for personalized recommendations, which may require adjustments due to updates in external APIs.
- **Development and Maintenance**
    - Use modern development tools such as Next.js, React, and Tailwind for frontend, and NestJS or Express for backend development.
    - Ensure maintainable codebase practices to support long-term sustainability and ease of updates.
- **Documentation and Support**
    - Provide comprehensive documentation for users and developers, including user manuals, installation guides, and API references.
- **Energy Efficiency**
    - Optimize for energy-efficient operation to minimize environmental impact and operational costs.

3.Use-Case Model



4.Logical View



The architecture diagram illustrates a system for managing music assets, playlists, and user interactions. It uses **Next.js** for the frontend, a combination of backend components built with **Express.js**, and a database setup combining **MongoDB** and **Cloudinary**. Below is a detailed explanation of each component and their connections:

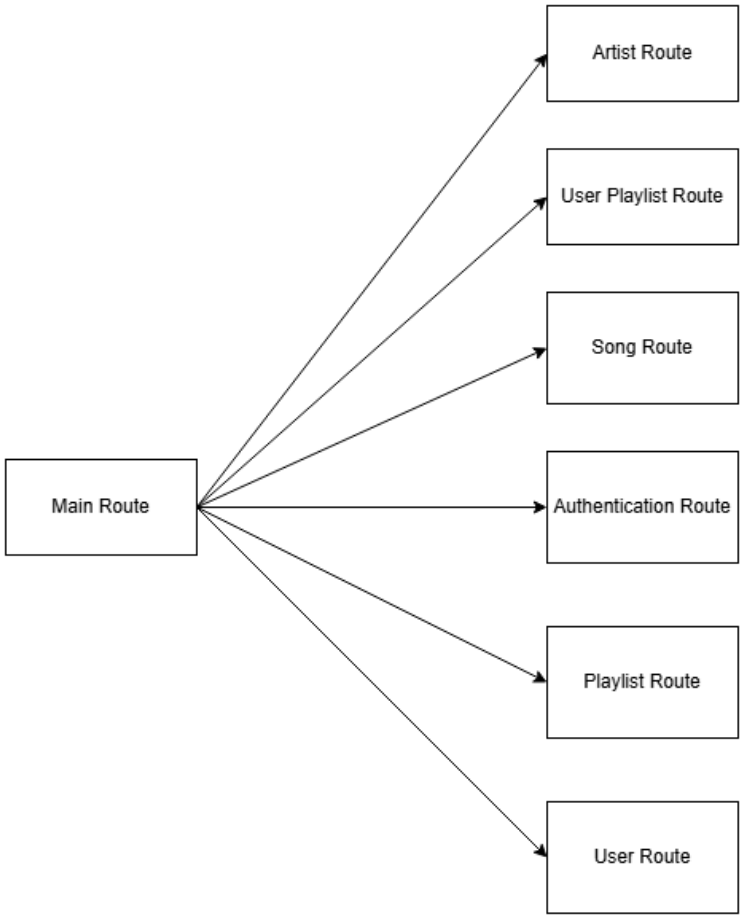
1. **Frontend:**
   - **Next.js:**
     - A React framework used to build server-side rendered and static web applications.
     - Provides built-in routing, server-side rendering, and optimizations for performance.
   - **View:**
     - The user interface layer where users interact with the system to explore playlists, songs, and artists.
     - Fetches data from the backend and dynamically renders the content.
   - **Tailwind CSS:**
     - A utility-first CSS framework for designing responsive and visually appealing user interfaces.
     - Ensures consistent and modern styling for the frontend components.
2. **Backend:**
   - **Routes:**
     - Define specific endpoints for handling requests from the frontend.
     - Organized into modules such as user routes, artist routes, playlist routes, and song routes.
     - Connects incoming HTTP requests to the appropriate controller for processing.
   - **Controllers:**
     - Implements the logic for handling requests routed from the frontend.
     - Performs actions such as user authentication, fetching playlists, uploading songs, and managing artist profiles.
     - Communicates with the models and Cloudinary to fetch or update data.
   - **Models:**
     - Represents the application's data structure and handles database interactions.
     - Stores metadata about users, playlists, songs, and artists in MongoDB.
     - Includes links to media files (audio, images) stored on Cloudinary.
3. **Database:**
   - **MongoDB:**
     - Serves as the primary NoSQL database for storing structured metadata.
     - Stores references (URLs) to media files hosted on Cloudinary.

- **Cloudinary:**
  - A media management platform for storing and delivering media assets.
  - Provides fast and secure content delivery via its CDN (Content Delivery Network).
  - Provides secure URLs for hosted media files, which are stored as references in MongoDB.
  - Used for storing and serving media assets such as song audio, artist profile images, and playlist cover art.

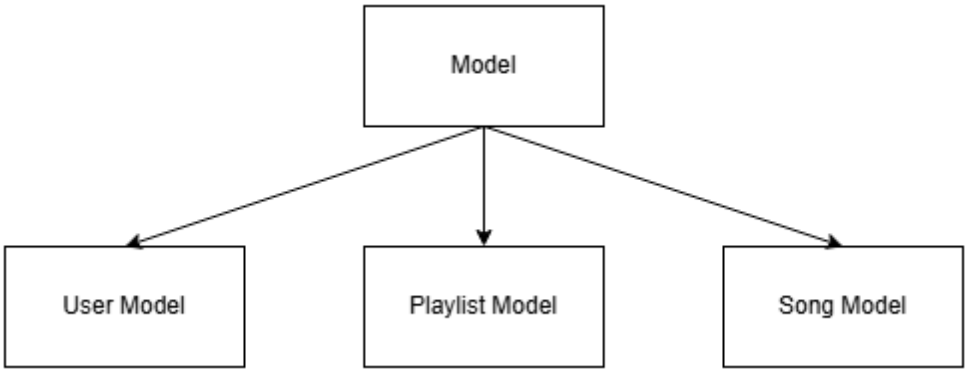
4. **Connections:**
   - **Frontend to Backend (Routes):**
     - The frontend communicates with backend routes over HTTPS to send requests or retrieve data.
     - Each route corresponds to a specific module for handling actions like fetching user data, playlists, or artist details.
   - **Backend to Database (MongoDB Query):**
     - The models perform CRUD (Create, Read, Update, Delete) operations on MongoDB to store and retrieve metadata about users, playlists, songs, and artists.
   - **Backend to Cloudinary (API):**
     - Media files (e.g., audio or images) are uploaded to Cloudinary using its API.
     - Cloudinary returns secure URLs, which are stored as metadata in MongoDB.

4.1. Component: Route

Descriptions: This component handles the incoming requests from the user. It determines which controller should handle a given request based on the URL or endpoint.

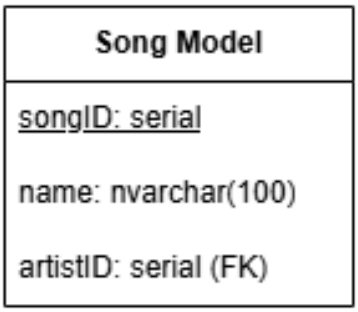


4.2. Component: Model



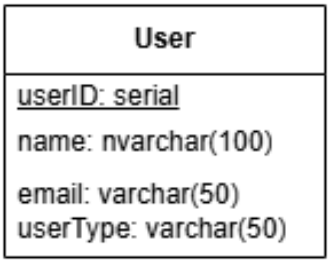
Description: The Model is the heart of the system. It handles the main logic, manages user data, playlists, and interacts with services like the Database, Streaming, and Recommendations. It makes sure everything runs smoothly and personalizes the music experience for users.

- **Song Model**



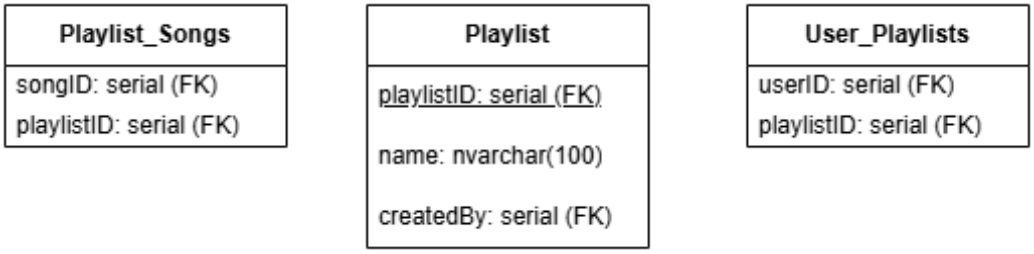
Description: The Song Model contains key data attributes such as songID, name, and artistID. The artistID serves as a foreign key, referencing the User Model, while the songID is the primary key for the Song Model.

- **User Model**



Description: The User Model includes key attributes such as userID, name, and email. The usertype is a string variable used to differentiate between different user roles, such as curator or admin.

- **Playlist Model**

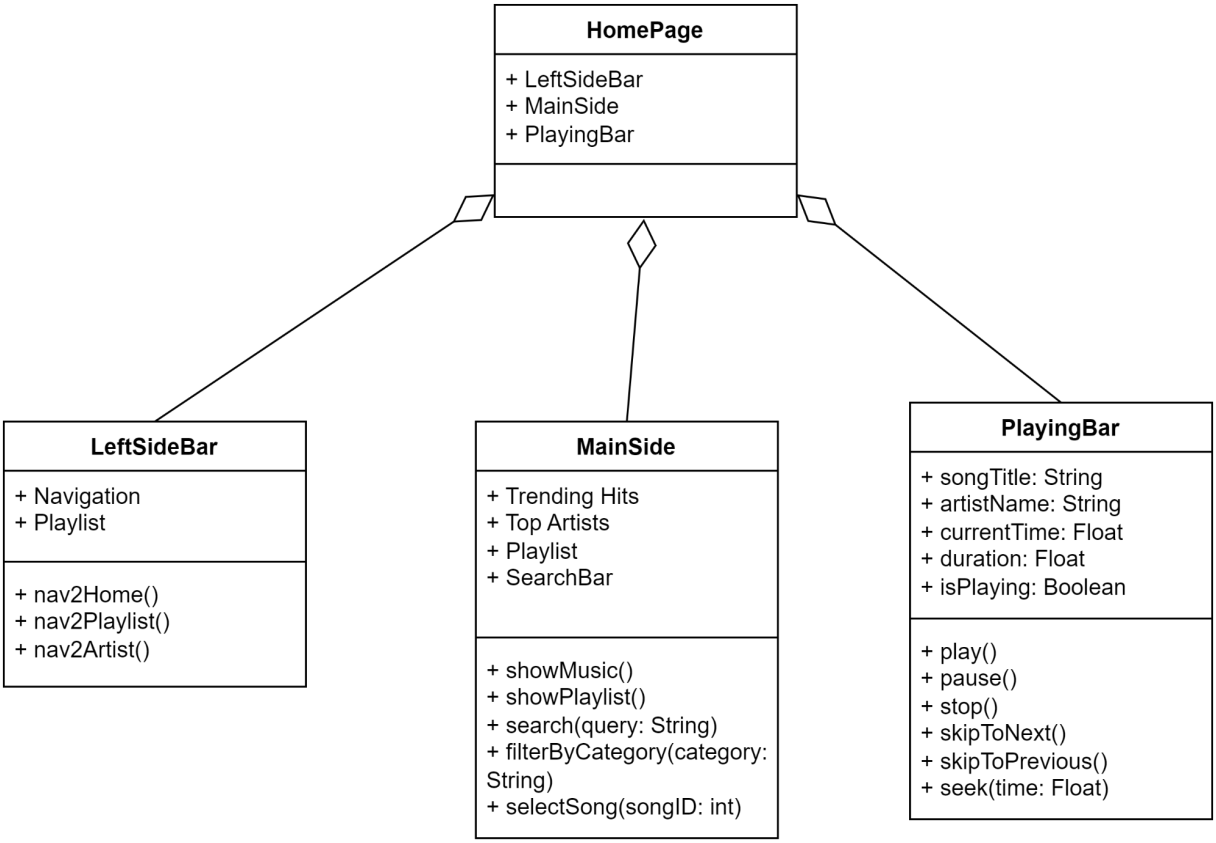


Description: The Playlist Model consists of three tables. The Playlist table stores primary data such as id and name. The Playlist_Songs table manages the songs within each playlist, while the User_Playlists table links playlists to individual users, managing which playlists belong to which users.

4.3. Component: View

Descriptions: The View is responsible for presenting data to the user. It generates the user interface (UI) by using the data provided by the Controller. In this product, we use only HTML, CSS and JavaScript, but no view engine or framework. So the View component is just HTML files which are sent by the server to the client. The View changes when the client calls an API from the server, then gets an on-page update, or when the client requests another page.

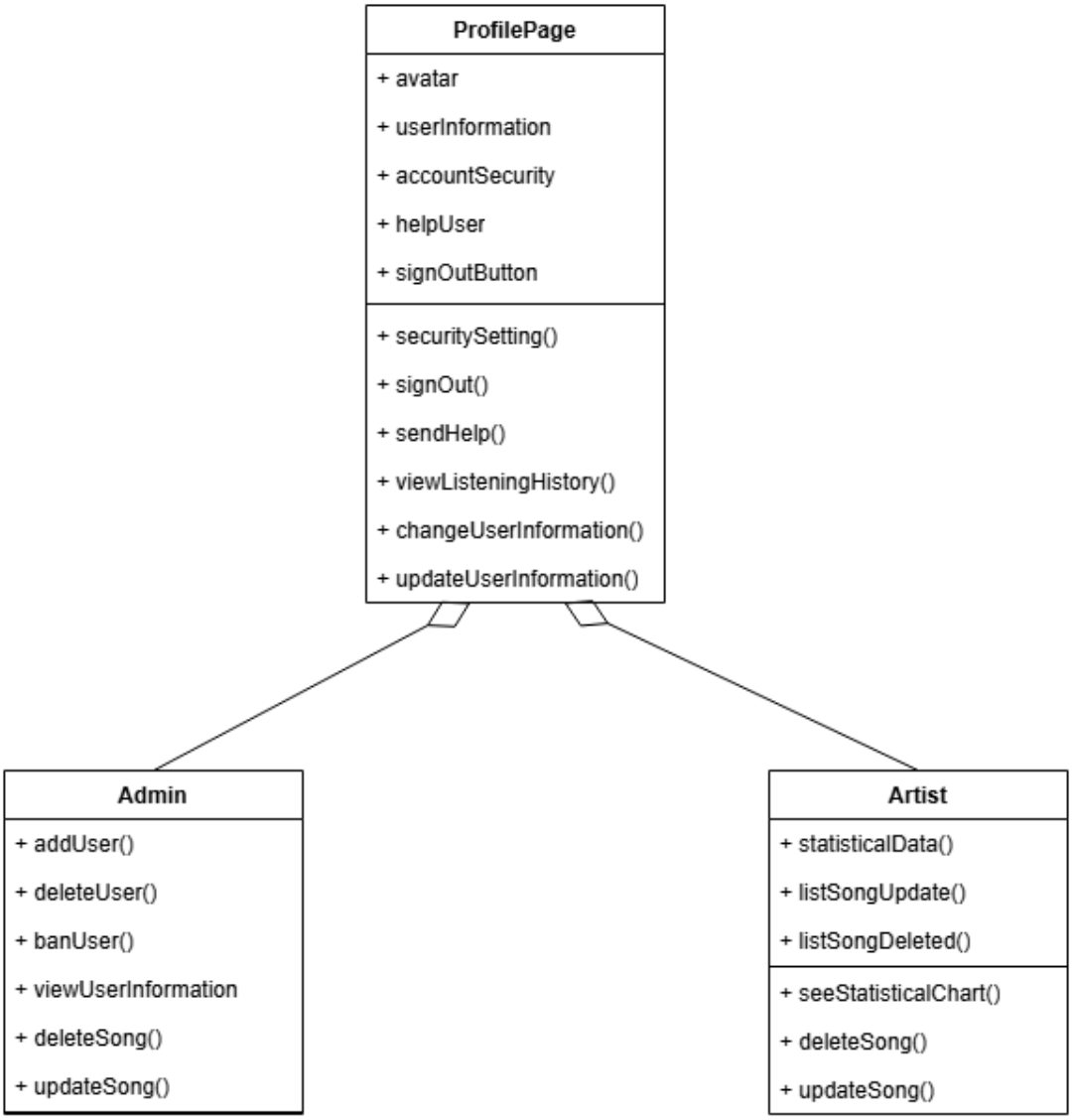
- **Home page**



Descriptions:

- **LeftSideBar**: A navigation area providing quick access to playlists, libraries, or other sections of the application.
- **MainSide**: The main content display area where users can browse and interact with media, such as songs or albums.
- **PlayingBar**: A bottom bar for playback controls, including play, pause, skip, and volume adjustments.

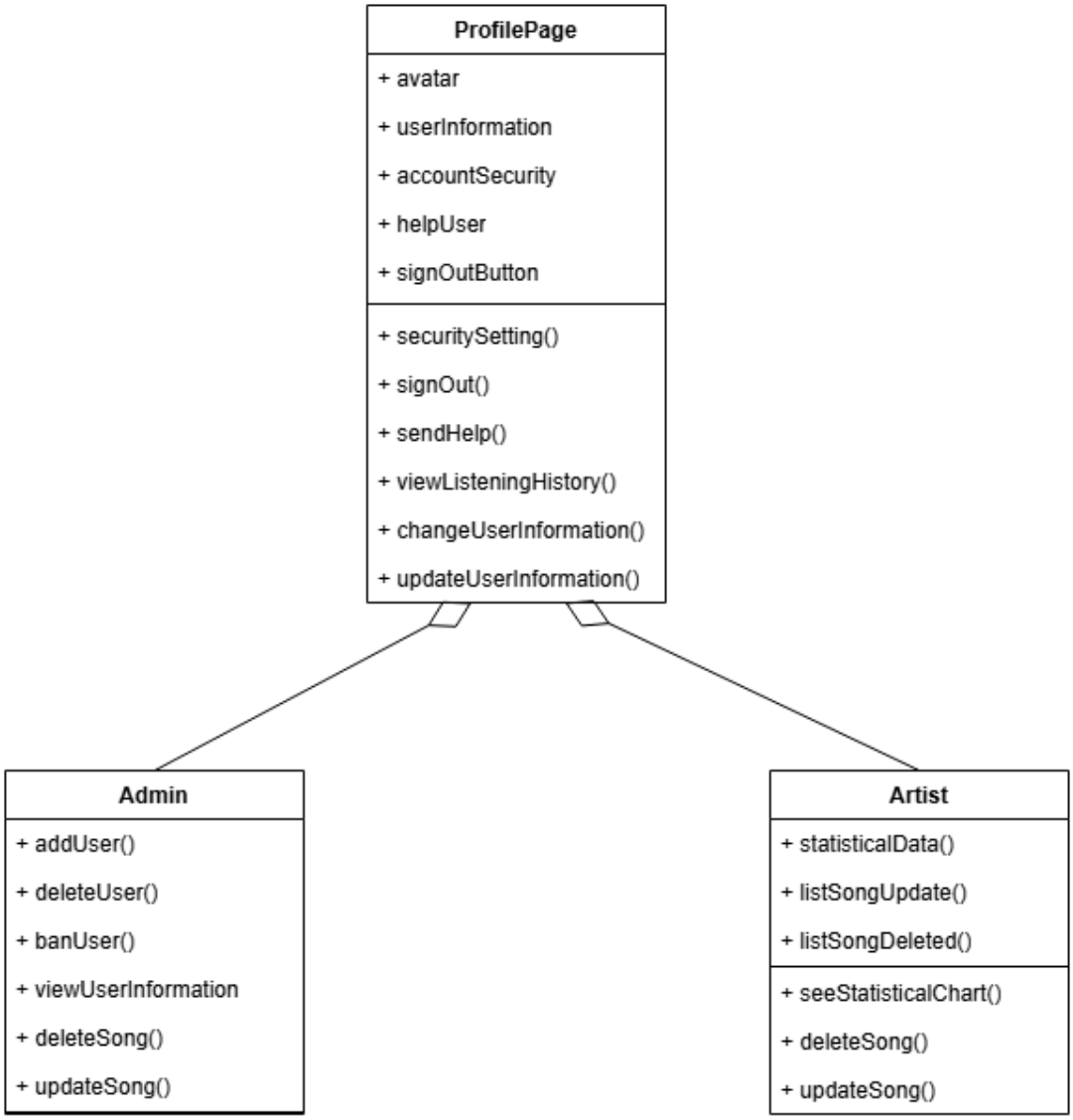
- **Login Page**



Descriptions:

- Only users who are logged in to the system can access features like viewing notifications and accessing profiles.
- Users can switch back and forth between the login and registration pages with just the click of a button.

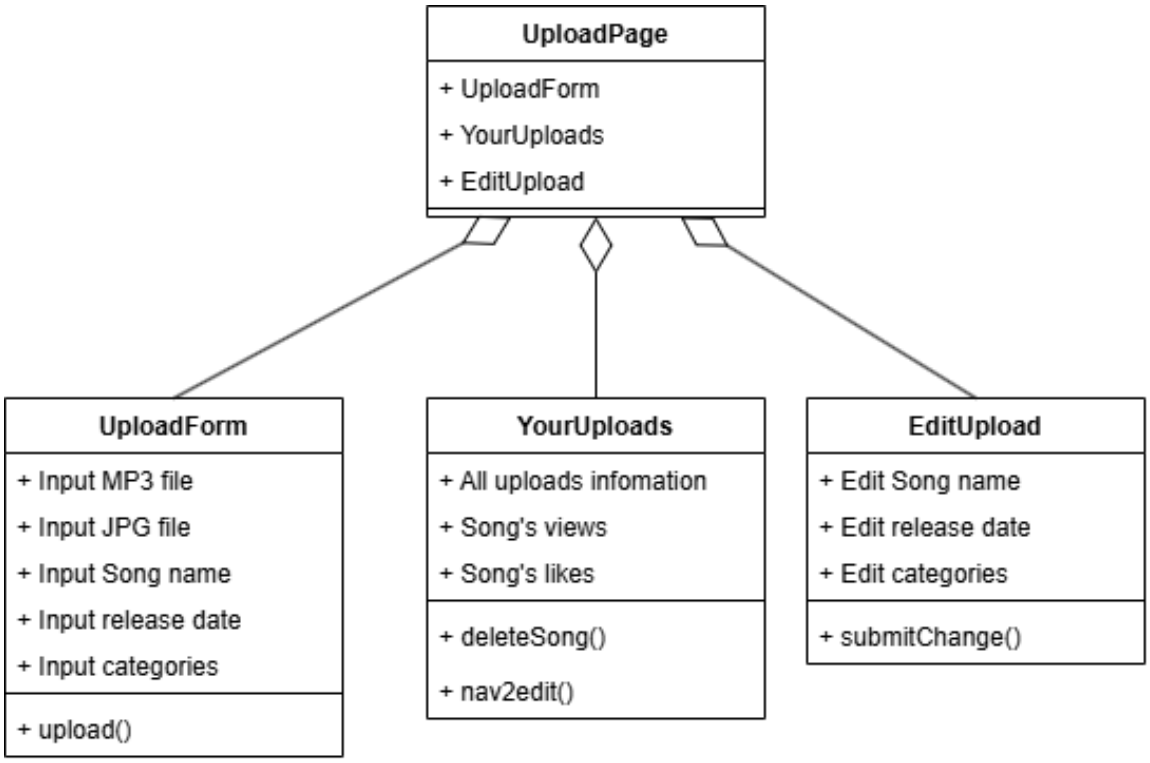
- **Profile Page**



Descriptions:

- Users can customize their information on their profile page.
- Users cannot upgrade accounts to admin accounts or artist accounts.
- Admin can view user information and manage users.
- Artists can upload music and view statistics related to their products such as views and most popular products.

- **UploadPage**



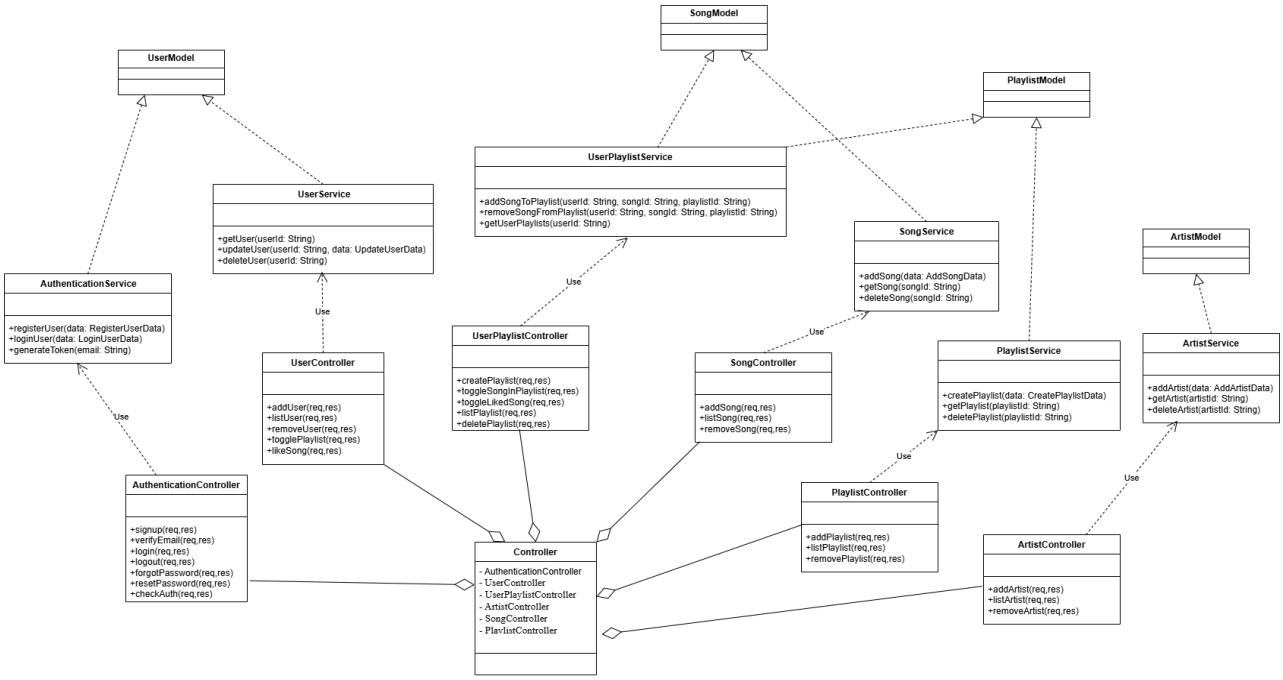
Descriptions:

- Only artists can access the upload page.
- The upload form allows users to upload new songs. Users can provide an MP3 file for the song, a JPG file for the cover image, the song's name, its release date, and categorize it. Once all information is entered, the form can be submitted to upload the song to the server.
- The YourUpload component displays a summary of all the songs uploaded by the user. It shows detailed information about each song, including the number of views and likes each song has received. Users can delete a song from their uploads or navigate to the edit page to make changes to an existing upload.
- The EditUpload component allows users to modify the details of their previously uploaded songs. Users can change the song's name, update the release date, and modify its categories. Once the changes are made, they can submit the form to update the song's information on the server.

4.4. Component: Controller

Controllers include:

1. AuthenticationController
2. UserController
3. UserPlaylistController
4. ArtistController
5. SongController
6. PlaylistController



Key-Class Explanation:

1. AuthenticationController

Manages various user-related functionalities, including registration, login, email verification, password reset, and session management. Below is a breakdown of each method.

Methods:

- **signup(req, res)**
  - **Purpose**: Registers a new user by creating a user account with email, password, and name, and sends an email verification code.
  - **Response**: Returns a success message with the user data (excluding password) if successful, or an error message otherwise.
- **verifyEmail(req, res)**
  - **Purpose**: Verifies a user's email using a provided verification code.
  - **Response**: Returns success and user data on successful verification, or an error message if the code is invalid or expired.
- **login(req, res)**
  - **Purpose**: Logs in a user with their email and password.
  - **Response**: Returns success and user data if credentials are valid, or an error message otherwise.
- **logout(req, res)**
  - **Purpose**: Logs out the user by clearing the authentication cookie.
  - **Response**: Returns a success message indicating logout.
- **forgotPassword(req, res)**
  - **Purpose**: Initiates the password reset process by sending a password reset email.
  - **Response**: Returns success if the email is sent, or an error message if the user is not found.
- **resetPassword(req, res)**
  - **Purpose**: Resets the user's password using a reset token.
  - **Response**: Returns success if the password is reset, or an error message if the token is invalid or expired.
- **checkAuth(req, res)**

- **Purpose**: Checks if the user is authenticated by finding them using `userId` from the request.
- **Response**: Returns the user data if found, or an error message if the user is not found.

2. UserController

Manages user operations, including adding new users, listing users, deleting users, and managing user playlists.

Methods:

- **addUser(req, res)**
- **Purpose**: Adds a new user with basic information, password, and profile picture.
- **Response**: Returns a success message with the new user details or an error message if an error occurs.
- **listUser(req, res)**
- **Purpose**: Lists all registered users.
- **Response**: Returns a list of users if successful, or an error message otherwise.
- **removeUser(req, res)**
- **Purpose**: Deletes a user by their ID.
- **Response**: Returns a success message if the user is deleted or an error message if an error occurs..
- **togglePlaylist(req, res)**
- **Purpose**: Adds or removes a song in a user's playlist.
- **Response**: Returns a success message with the updated playlist or an error message otherwise.
- **likeSong(req, res)**
- **Purpose**: Adds or removes a song from the user's "Liked Songs" playlist.
- **Response**: Returns a message indicating whether the song was added or removed.

3. UserPlaylistController

Handles playlist-related operations, such as creating playlists, adding/removing songs, and managing a user's "Liked Songs" playlist.

Methods:

- **createPlaylist(req, res)**
- **Purpose**: Creates a new playlist for a specified user.
- **Response**: Returns the newly created playlist or an error message.
- **toggleSongInPlaylist(req, res)**
- **Purpose**: Adds or removes a song in a specified playlist.
- **Response**: Returns a success message and the updated playlist or an error message if unsuccessful.
- **toggleLikedSong(req, res)**
- **Purpose**: Adds or removes a song in the user's "Liked Songs" playlist.
- **Response**: Returns a success message and the updated playlist or an error message if unsuccessful.
- **listPlaylists(req, res)**
- **Purpose**: Lists all playlists for a specific user.
- **Response**: Returns the list of playlists or an error message if unsuccessful.
- **deletePlaylist(req, res)**
- **Purpose**: Deletes a specified playlist.
- **Response**: Returns a success message or an error message if unsuccessful.

4. ArtistController

Handles artist-related operations, such as add artist, remove artist and list artist.

Methods:

- **addArtist(req, res)**
- **Purpose**: Adds a new artist to the database with their name, description, background color, and profile image.
- **Response**: Returns a success message if the Artist is created, or an error message if an error

occurs.

- **listArtist(req, res)**
- **Purpose**: Retrieves a list of all artists from the database.
- **Response**: Returns a success message along with the list of all artists if the retrieval is successful, or an error message otherwise.
- **removeArtist(req, res)**
- **Purpose**: Deletes an artist from the database by their ID.
- **Response**: Returns a success message if the artist is deleted, or an error message if an error occurs.

5. SongController

Manages sonsong-related operations such as adding a song, listing all songs, and deleting a song

Methods:

- **addSong(req, res)**
- **Purpose:** Adds a new song with a name, description, background color, and an image.
- **Response**: Returns a success message if the song is created, or an error message if an error occurs.
- **listSong(req, res)**
- **Purpose**: Retrieves and returns a list of all songs in the database.
- **Response**: Returns a success message along with the list of songs if the retrieval is successful, or an error message otherwise.
- **removeSong(req, res)**
- **Purpose**: Deletes a playlist by its ID.
- **Response**: Returns a success message if the song is deleted, or an error message if an error occurs.
-

6. PlaylistController

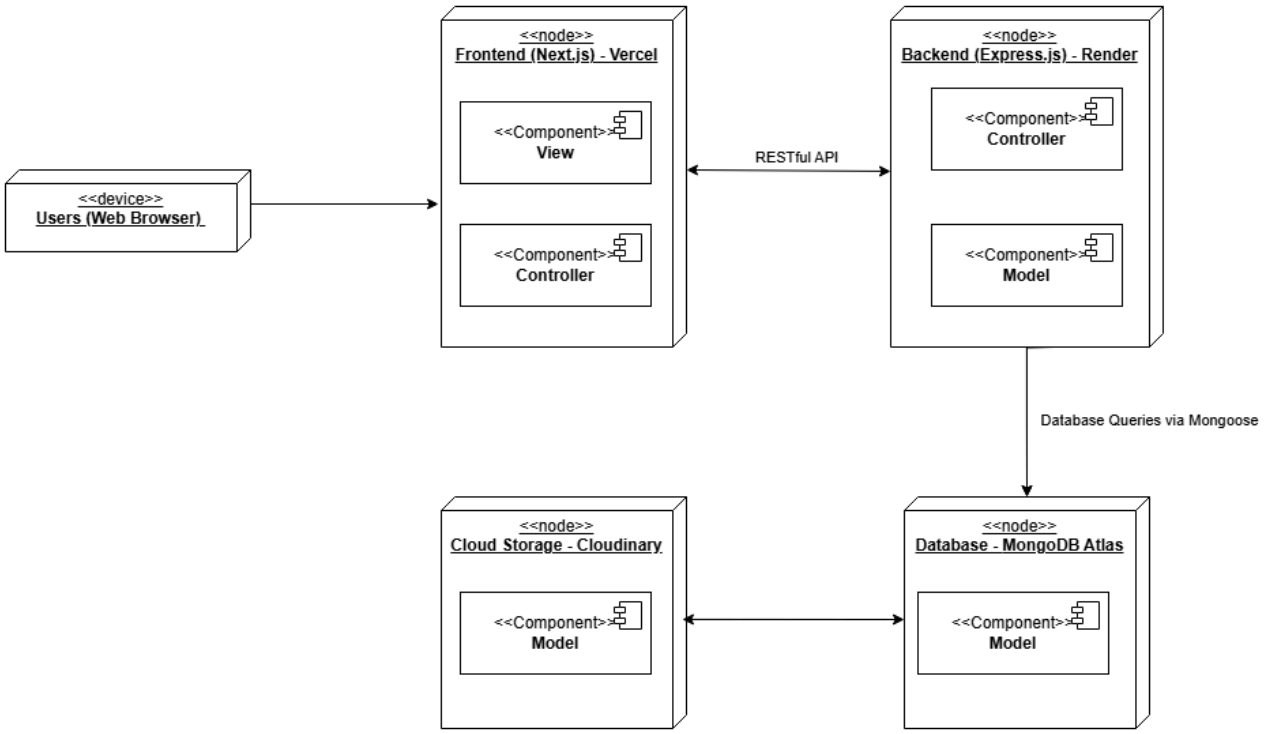
Manages playlist-related operations such as adding a playlist, listing all playlists, and deleting a playlist

Methods:

- **addPlaylist(req, res)**
- **Purpose:** Adds a new playlist with a name, description, background color, and an image.
- **Response**: Returns a success message if the playlist is created, or an error message if an error occurs.
- **listPlaylist(req, res)**
- **Purpose**: Retrieves and returns a list of all playlists in the database.
- **Response**: Returns a success message along with the list of playlists if the retrieval is successful, or an error message otherwise.
- **removePlaylist(req, res)**
- **Purpose**: Deletes a playlist by its ID.
- **Response**: Returns a success message if the playlist is deleted, or an error message if an error occurs.

5. Deployment

The deployment of **Beatify**, our music streaming platform, involves distributing and running the various system components across different machines and services. Below is a mapping of each component to its respective deployment environment, including details about the hosting services and how each part of the system interacts with the overall architecture.



1. Frontend (View) - Next.js Application

The frontend of Beatify is built using the **Next.js** framework, providing both static and dynamic rendering capabilities for a seamless user experience. The application is responsible for handling the user interface, displaying music content, managing playlists, and interacting with the backend API.

- **Deployment Environment**: The frontend is hosted on **Vercel**, a cloud platform optimized for Next.js applications. Vercel automatically optimizes performance with features like server-side rendering (SSR) and edge caching, ensuring that users can access the app quickly regardless of their geographical location.
- **Components**:
    - **View**: Handles the presentation logic (HTML, CSS, and JavaScript) and communicates with the backend via APIs (RESTful or GraphQL).
    - **Controller**: In the frontend, this is typically the part of the app that handles user input and makes calls to the backend APIs (e.g., user actions to play a song or manage a playlist).

2. Backend (Controller and Model) - Express.js Application

The backend is built using **Express.js**, a minimal and flexible Node.js web application framework. It serves as the intermediary between the frontend and the database, processing user requests, managing authentication, and serving dynamic content such as audio streams, metadata, and user-specific information.

- **Deployment Environment**: The backend is deployed to **Render**, a cloud platform designed for deploying full-stack applications. Render provides easy scaling and management of services like databases, backend applications, and APIs, making it an ideal choice for hosting the Express.js server.
- **Components**:
    - **Controller**: Manages incoming HTTP requests, such as GET requests for song information, POST requests for user data, etc. It routes requests to the appropriate service or model for further processing.

- **Model**: Interacts with the database, such as querying MongoDB for user data, playlists, and music content, and managing the business logic behind user interactions. MongoDB Atlas is used as the database system, providing a fully managed NoSQL database that scales automatically to meet application demand.

3. Database - MongoDB Atlas

Beatify stores user data, music metadata, and playlist information in a **MongoDB Atlas** cluster, which is a managed, cloud-based version of MongoDB. MongoDB Atlas provides high availability, automatic backups, and scalability to ensure the application can handle large amounts of user and content data.

- **Deployment Environment**: The MongoDB Atlas service is hosted in the cloud, with automatic replication and sharding across multiple regions to ensure the database is highly available and performant.
- **Components**:
  - **Model**: The backend (Express.js) communicates with MongoDB Atlas to store and retrieve data. This includes user profiles, song metadata, playlists, and more.

4. Cloud Storage - Cloudinary

Beatify relies on **Cloudinary** for storing and serving media files such as images (album covers) and audio files (songs). Cloudinary is a cloud service that handles image and video uploads, storage, and dynamic transformations (e.g., resizing or format conversion), and also serves audio content to users in an efficient and scalable manner.

- **Deployment Environment**: Cloudinary is a fully managed cloud service that hosts media files. Beatify interacts with Cloudinary via their REST API to upload and fetch media assets.
- **Components**:
  - **Model**: The backend communicates with Cloudinary to upload, retrieve, and manage media files, such as song tracks, album covers, or user profile images.

6.Implementation View

Backend Folder Structure:

```
backend/
├── mailtrap/              # Folder for managing email services using Mailtrap
├── node_modules/          # Auto-generated dependencies for Node.js
├── src/                   # Source code for the backend
│   ├── config/            # Configuration files (e.g., environment variables)
│   ├── controllers/       # Contains logic for handling API requests
│   ├── middleware/        # Custom middleware functions for the Express.js app
│   ├── models/            # Defines MongoDB schemas and data models
│   ├── routes/            # API routes to handle various endpoints
│   ├── utils/             # Utility functions used across the backend
├── .env                   # Environment variables (e.g., API keys, database URIs)
├── .gitignore             # Specifies files and directories ignored by Git
├── package-lock.json      # Dependency lock file for consistent installations
├── package.json           # Manages project dependencies and scripts
├── product.js             # Module for product-related logic (example file)
└── server.js              # Entry point of the Express.js server
```

The backend is implemented using Node.js and Express.js with a focus on modular and feature-based organization.

Key Highlights

- **Controllers:** Handle the core API request/response logic (e.g., /products, /users).
- **Middleware:** Reusable Express.js middlewares like logging, authentication, etc.
- **Models:** MongoDB schemas using **Mongoose** for consistent data representation.
- **Routes:** Define API routes for handling endpoints (e.g., routes/product.js).
- **Utils:** Shared utility functions like date handling or error formatting.

Frontend Folder Structure:
The frontend leverages Next.js for server-side rendering, improved routing, and modular UI development. It also integrates Tailwind CSS for styling.

```
frontend/
├── .next/                  # Auto-generated Next.js build files
├── node_modules/           # Auto-generated dependencies for Node.js
├── public/                 # Public assets (e.g., static images and icons)
├── src/                    # Source code for the frontend
│   ├── app/                # Application structure for Next.js routes
│   │   ├── (admin)/        # Admin-specific pages and components
│   │   ├── (auth)/         # Authentication-related pages and components
│   │   ├── (site)/         # Main user-facing pages and components
│   ├── assets/             # Static assets like images or fonts
│   ├── components/         # Reusable UI components (e.g., buttons, modals)
│   ├── contexts/           # React context providers for global state management
│   ├── store/              # State management (e.g., Redux or Zustand setup)
│   ├── utils/              # Utility functions used across the frontend
├── .eslint.json            # Configuration file for ESLint
├── .gitignore              # Specifies files and directories ignored by Git
├── next.config.js          # Next.js configuration settings
├── package-lock.json       # Dependency lock file for consistent installations
├── package.json            # Manages project dependencies and scripts
├── README.md               # Documentation for the project
├── tailwind.config.ts      # Tailwind CSS configuration file
└── tsconfig.json           # TypeScript configuration for the project
```

Key Highlights

1. **Pages:**
   - **(auth):** Routes for login, registration, and password management.
   - **(site):** Main user pages such as home, product details, and listings.
   - **(admin):** Admin dashboard and tools for managing resources.
2. **Components:**
   - Reusable components like buttons, cards, and modals.
3. **State Management:**
   - store/: Redux/Zustand setup for managing global application state.
4. **Styling:**
   - Tailwind CSS for utility-first styling and global CSS **(globals.css).**
5. **API Integration:**
   - Use **src/lib** for organizing backend interactions, utilities, and logic.