**Chapter - 1**

**SYNOPSIS**

* 1. **Title of the project:** **Banger**
  2. **Objective of the project**:

The objective of Banger is to provide users with a seamless streaming experience while integrating social interaction through real-time chat. Users can log in, listen to songs, and see what others are currently playing, enhancing engagement and discovery.

* 1. **Problem analysis:**

Banger is smooth and user-friendly music player site. As compare to Current music player doesn’t have Real time chat system with All users(social-site)and it have new feature that user can see what others are listening.

**1.4 Modules:**

The system consists of Three modules – Administrative module, User module, Super admin module

**Administration Module:**

✅ **Song Management**

* Add new songs with details like title, artist, album, and duration.
* Edit existing songs to update metadata or replace files.
* Delete songs to maintain a clean and relevant library.

✅ **Album Management**

* Add new albums with album covers, descriptions, and related songs.
* Edit album details, including updating song lists and cover images.
* Remove outdated or irrelevant albums

✅ **Dashboard Overview**

* View total **songs**, **artists**, **users**, and **albums** at a glance.

**User Modules:**

✅ **Listen to Songs & Albums**

Stream songs individually or browse through albums for a curated experience.

✅ **Volume Control & Playback**

Adjust volume levels and control playback with play, pause, next, and previous options.

✅ **Search for Songs & Albums**

Quickly find specific songs or albums using a powerful search function.

✅ **Profile Management**

Update user details, including changing passwords and managing account settings.

✅ **Live Listening Activity**

See what other users are currently listening to in real time.

✅ **Real-Time Chat**

Chat with other users instantly, just like on social media.

Engage in music discussions and share recommendations.

**Super Admin Module:**

🟢 **User Management** – View, manage, and delete user accounts.

🟢 **Role & Permission Control** – Assign admin, user, and moderator roles.

🟢 **Authentication & Security** – Manage login methods (email, social login, etc.).

🟢 **Session Management** – Monitor active user sessions and log out users if needed.

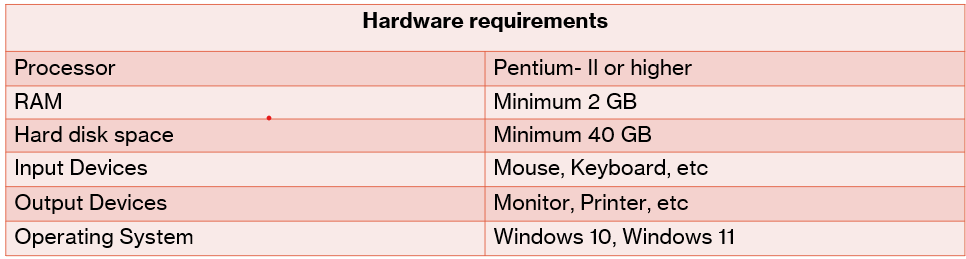
🟢 **Activity Logs** – Track user activity and system interactions.

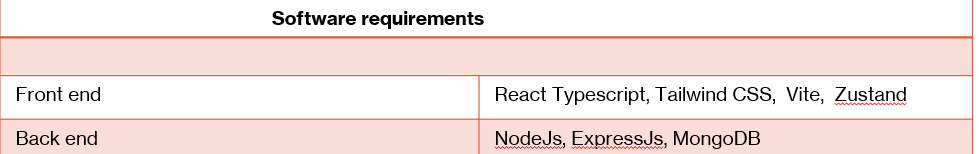
🟢 **Custom Fields & Metadata** – Store and manage additional user data.

🟢 **Audit & Compliance** – Ensure platform security and compliance with policies.

* 1. **Future Scope:**
* **Mobile Application Development** – Expand Banger to portable devices like smartphones and tablets for seamless music streaming on the go.
* **AI-Based Music Recommendations** – Implement AI-driven personalized song and album recommendations based on user preferences and listening history.
* **Live Streaming & Artist Interaction** – Introduce live sessions where artists can interact with users and stream exclusive content.
* **Voice Commands & Smart Assistant Integration** – Allow users to control playback and search for songs using voice commands.

**1.6 Hardware and Software Requirements:**





**Chapter – 2**

**LITERATURE SURVEY**

**2.1 React TS + Vite:**

* **React + TypeScript** is a powerful combination for building modern, scalable web applications. React provides the UI framework, while TypeScript adds static typing to JavaScript, improving code quality, maintainability, and developer experience.
* **React**: A JavaScript library for building user interfaces, React allows developers to create dynamic and responsive UIs using components. It's widely used for single-page applications (SPAs).
* **TypeScript**: TypeScript is a superset of JavaScript that introduces static typing. It helps developers catch errors at compile-time, making the codebase more robust, especially in large-scale applications. TypeScript enhances code readability and simplifies refactoring.
* **Vite** is a next-generation build tool designed to improve the development experience. It's extremely fast and lightweight, providing better performance and features compared to older tools like Webpack.
* Together, **ReactTS** and **Vite** offer:
* **Fast Development Build**: Vite’s super-fast hot reloading makes the development process highly efficient.
* **Improved Code Quality**: TypeScript adds strong typing, preventing many common errors and enhancing the developer experience.
* **Optimized Production Build**: Vite produces highly optimized code for production, improving performance in production environments.

**2.2 Tailwind CSS:**

* **Tailwind CSS** is a utility-first CSS framework that provides a highly customizable and efficient way to design web applications. Unlike traditional CSS frameworks (e.g., Bootstrap), which offer pre-defined components and styles, Tailwind provides low-level utility classes that you can combine to create custom designs.
* **Highly Customizable**
* Allows customization of colors, spacing, typography, etc., via a configuration file.
* **Responsive Design**
* Built-in responsive utilities for different screen sizes (e.g., sm:, md:, lg:).
* **Performance Optimization**
* Integrates with **PurgeCSS** to remove unused CSS, reducing file size.
* **Consistency & Reusability**
* Promotes consistent design across the app and reusability of utility classes.
* **Faster Development**
* Speeds up development by using utility classes instead of custom CSS.
* **Perfect for Prototyping**
* Ideal for quickly creating layouts and designs without writing a lot of custom CSS.

**2.3 Node & Express:**

1. **Node.js**

Node.js is an open-source, cross-platform runtime environment that allows you to execute JavaScript code on the server-side. Unlike traditional JavaScript running in the browser, Node.js enables JavaScript to be used for backend development, making it possible to build fast and scalable web servers.

**Non-blocking I/O**: Node.js is designed to handle asynchronous operations, such as reading from files or handling database queries, without blocking the rest of the application.

**Single-threaded Event Loop**: Node.js uses a single thread to handle multiple requests using an event-driven, non-blocking I/O model, which helps in handling concurrent connections efficiently.

**2. Express.js**

 is a lightweight, flexible framework built on top of Node.js that simplifies the process of building web applications and APIs. It provides a robust set of features for creating server-side applications, such as routing, middleware support, and request handling.

* **Simplified Routing**: Express allows developers to define routes easily using HTTP methods (GET, POST, PUT, DELETE) and URL paths.

**2.4 MongoDB:**

* **MongoDB** is a popular, open-source, NoSQL database that uses a document-oriented data model. It is designed to handle large amounts of unstructured data and provides high performance, flexibility, and scalability for modern web applications.
* **Document-Oriented**: Stores data as flexible, JSON-like documents (BSON) which can contain nested data structures.
* **Scalability**: Supports horizontal scaling (sharding) across multiple servers to handle large data volumes and high traffic.
* **Schema Flexibility**: No fixed schema; allows documents within the same collection to have different fields and structures.
* **High Availability**: Data replication across multiple servers ensures system availability and failover in case of server failure.
* **Indexing**: Supports various types of indexes to optimize query performance and speed up data retrieval.
* **Aggregation Framework**: Provides a powerful framework for performing data transformations, grouping, and computations.

**Chapter - 3**

**SOFTWARE REQUIREMENTS SPECIFICATION**

**3.1 Introduction:**

* A **Software Requirements Specification (SRS)** document is a detailed description of the software's intended functionalities and features, outlining what the software is supposed to accomplish without delving into the specific methods or technologies that will be used to achieve it.
* It focuses on defining the external behavior and expectations of the system, providing a clear understanding of its goals and requirements.

**3.2 Purpose:**

* The **purpose of this SRS** is to define the requirements for the web-based software application clearly.
* It serves as a communication tool, bridging the gap between developers and end users.
* Additionally, the SRS helps users better understand their own needs and expectations from the software.
* The ultimate goal is to create a high-quality, stable, and well-defined Software Requirements Specification

**3.3 Scope:**

* This document offers a thorough description of the functionality of **Banger**.
* It details the system's requirements, including its interfaces and interactions.
* Any modifications to the requirements in the future will follow a formal change approval procedure.
* The document is intended for developers and will serve as the basis for validating the final system upon delivery. This SRS comprehensively defines the requirements for **Banger**.

**3.4 Overview:**

This SRS provides a detailed understanding of what is expected from **Banger**. It will guide the development of future stages of the project, ensuring clarity in its features and functionalities

**3.5. Overall Description:**

**3.5.1 Product Perspective:**

Banger allows users to stream music, create playlists, interact with others in real-time, and manage their profiles. It supports features such as song and album search, volume control, and displaying what other users are listening to. Additionally, it includes an admin dashboard for managing the songs and albums, enabling easy updates and modifications. The system is designed to be intuitive, offering a seamless experience for both users and administrators.

**3.5.2 Product Future:**

* The product has the following features:
* **Banger** is an advanced web application that allows users to stream music, create playlists, and interact with other users in real-time.
* Only authenticated users can log in to the system, ensuring secure access to personalized features.
* The platform provides a music discovery experience, where users can search for songs and albums and listen to them based on their preferences.
* **Banger** has two user interface layers: one for regular users and another for administrators, offering distinct functionalities for each.
* Users must authenticate using their login credentials (ID and password) to access their accounts.
* The administrator has full control over the platform, including the ability to manage the music library, user accounts, and monitor system usage. Administrators can add, edit, and delete songs and albums.

**3.5.3 User Classes and Characteristics:**

* **Users**: Regular users who stream music, create playlists, interact with other users in real-time, and manage their profiles. They can listen to songs, search for albums, and see what others are listening to.
* **Administrators**: Responsible for managing the **Banger** platform. They have full access to the system and can manage the music library, add or remove songs and albums, and oversee user activity. Administrators ensure the system runs smoothly and that all content is up-to-date.

**3.5.4 General Constraints:**

* The developed system should be compatible with all major platforms (Windows, Mac, Linux, UNIX, etc.) that support modern web browsers.
* Internet connectivity is required for users to access and use the platform.
* Only authorized users who have registered and logged in can access the system's features and functionalities.

**3.5.5 Assumptions and Dependencies:**

* Users should have basic computer literacy and familiarity with navigating web applications.
* Users may need to be trained on how to utilize the system's features effectively.
* Some information will need to be manually entered by users, as it cannot be automatically generated by the system.
* The administrator role has already been created within the system.
* Roles and tasks for both users and administrators are predefined and managed within the system.

**3.6 Specific Requirements:**

**User interfaces-**

Each part of the user interface intends to be as user friendly as possible. The fonts and buttons used will be intended to be very fast and easy to load on web pages. The pages will be kept light in space so that it won’t take a long time for the page to load.

**Software interfaces-**

Operating System: Unix, Linux, Mac, Windows etc.

Development tool: Nodejs and Express js

Database: MongoDB

**External interfaces-**

Processor: Pentium or Higher

RAM: Minimum 2GB or above

**3.7 Design constraints:**

* Banger does not include a traditional login form. The only login option available is through Google authentication, allowing users to sign in seamlessly with their Google account.
* All user inputs, including during registration and profile updates, must be validated. If the data is invalid, error messages will be displayed, and the incorrect data will be ignored.
* The registration and user details are stored securely in the database, including information obtained via the Google authentication process.

**3.8 Security constraint:**

* **Banger** uses **MongoDB** for database management and storage, ensuring secure data handling, particularly with user profiles, playlists, and authentication details.
* The **administrator** has full rights to manage the software, including overseeing the music library, user data, and content updates.

**Chapter – 4  
SYSTEM DESIGN**

**4.1 Introduction:**

**System Design** is the process of planning how a software system will work. It includes deciding how data flows, how users interact, and how different parts of the system connect and function together.

* Architecture Design
* High Level Design
* Detailed Design

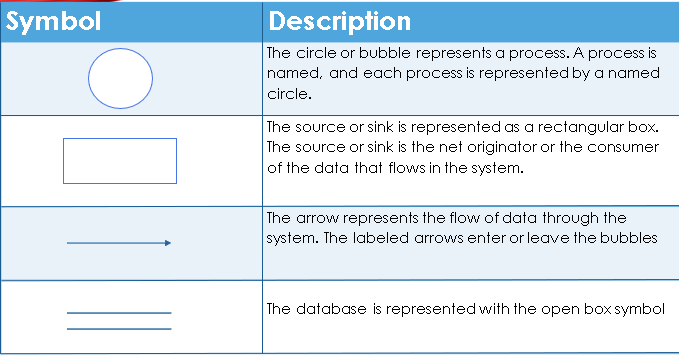
SYSTEM CONTEXT DIAGRAM:

* The System Context Diagram or Context Flow Diagram
* CFD is a top-level dataflow diagram. The System Context Diagram or Context Flow Diagram (CFD) describes the external entities acting on the system. In context diagram the entire system is treated as a single process with external behavior of software.

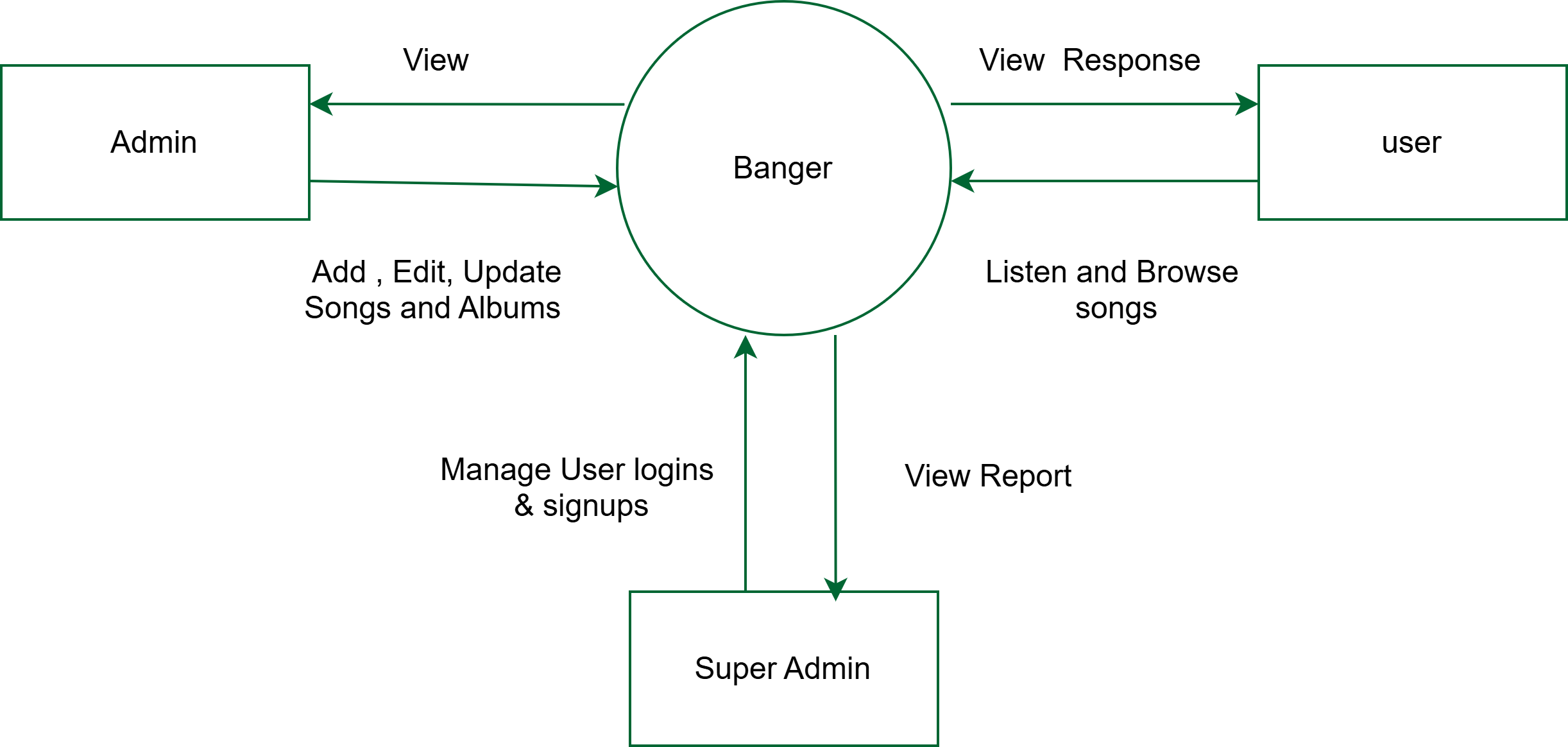
**4.2 Data Flow Diagram**:

A **Data Flow Diagram (DFD)** is a graphical representation of the “flow” of the data through an information System. It is common practice for designer to draw a context level DFD first which shows the interaction between the system and outside entities.

Four simple notations are used to complete a DFD. These notations are given below: -

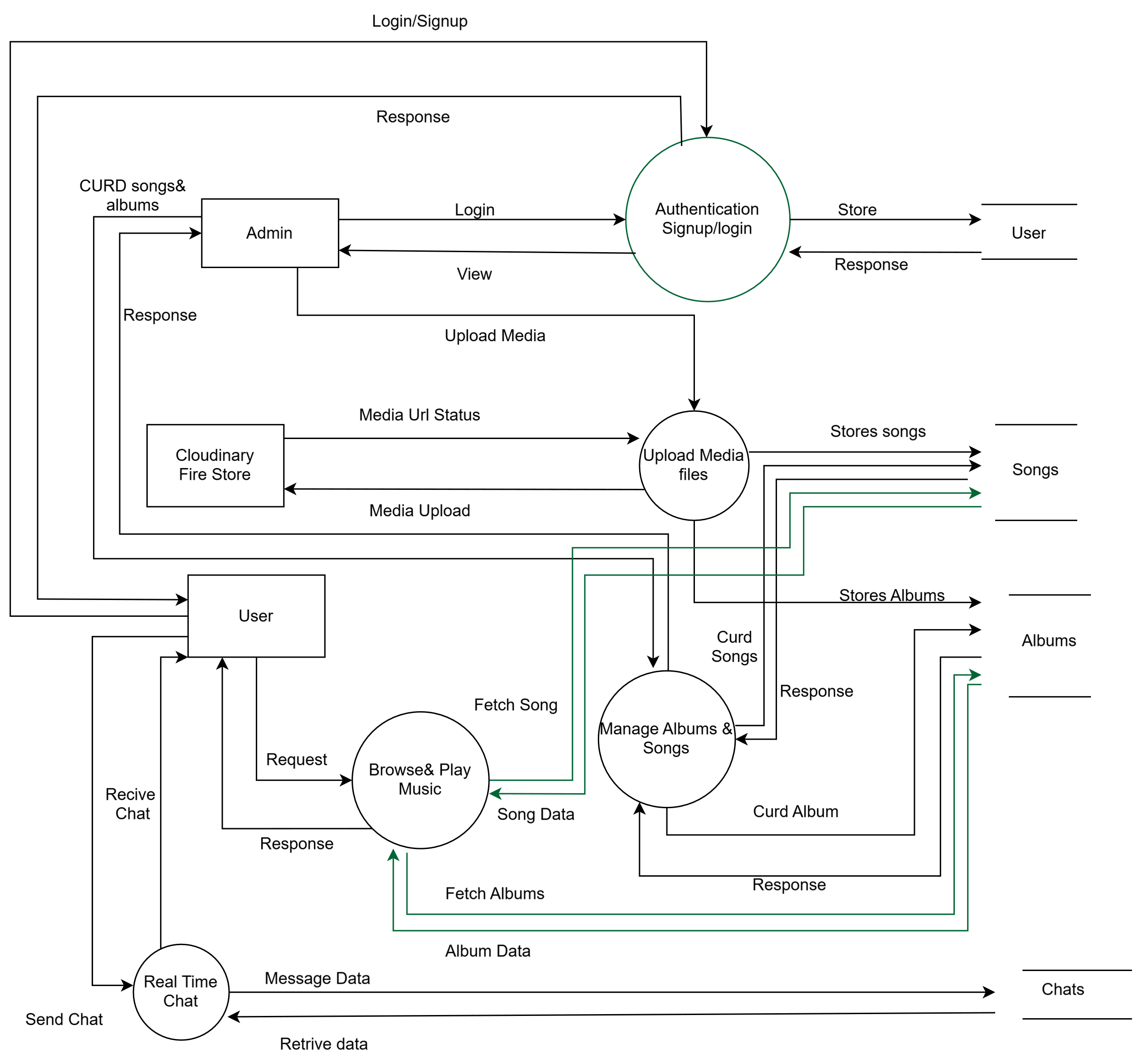


**4.3 Context Diagram:**



**4.4 Rules for Constructing DFD:**

* Process should be named for easy understanding.
* The direction of flow, top to bottom and from left to right should be specified.
* The direction of flow should not allow any kind of loops

**4.5 DFD Level-1:**

**4.6 DFD Level-2 User**:

A diagram of a user

AI-generated content may be incorrect.

**4.7 DFD Level-2 Admin:-**

A diagram of a curd album

AI-generated content may be incorrect.

**Chapter – 5  
 DATABASE DESIGN**

**5.1 Introduction:**

* **Database:** A Database is collection of related data, which can be of any size and complexity. By using the concept of Database, we can easily store and retrieve the data.
* **Database Design:** Database design is a process of organizing the data in an orderly manner. Database Design maintains the data required by the system.

**Schema Description User:**

|  |  |  |  |
| --- | --- | --- | --- |
| Field Name | Type | Constraints | Description |
| \_id | ObjectId | Primary key | Auto-generated User ID |
| fullName | String | Required | Full name of the user |
| imageUrl | String | Required | URL to user's image |
| clerkId | String | Required, Unique | Unique Clerk ID |
| createdAt | Date | Auto-generated (timestamp) | Record creation time |
| updatedAt | Date | Auto-generated (timestamp) | Record update time |

**5.2 Message:**

|  |  |  |  |
| --- | --- | --- | --- |
| Field Name | Type | Constraints | Description |
| \_id | ObjectId | Primary key | Auto-generated Message ID |
| senderId | String | Required | Clerk ID of the sender |
| receiverId | String | Required | Clerk ID of the receiver |
| content | String | Required | Message content |
| createdAt | Date | Auto-generated (timestamp) | Message creation time |
| updatedAt | Date | Auto-generated (timestamp) | Message last update time |

**5.3 Songs:**

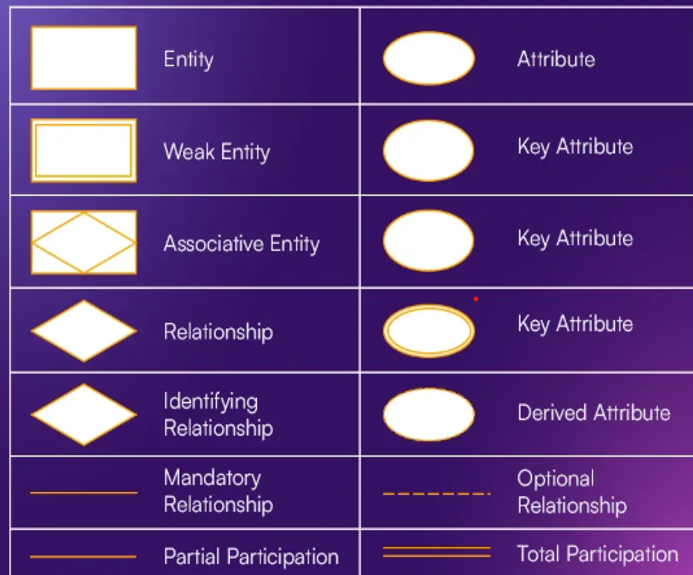
|  |  |  |  |
| --- | --- | --- | --- |
| Field Name | Type | Constraints | Description |
| \_id | ObjectId | Primary key | Auto-generated Song ID |
| title | String | Required | Title of the song |
| artist | String | Required | Artist name |
| imageUrl | String | Required | URL to the song image/cover |
| audioUrl | String | Required | URL to the song audio file |
| duration | Number | Required | Duration of the song (in seconds) |
| albumId | ObjectId (ref) | Optional (references Album) | Linked album ID |
| createdAt | Date | Auto-generated (timestamp) | Song creation time |
| updatedAt | Date | Auto-generated (timestamp) | Song update time |

**5.4 Album:**

|  |  |  |  |
| --- | --- | --- | --- |
| Field Name | Type | Constraints | Description |
| \_id | ObjectId | Primary key | Auto-generated Album ID |
| title | String | Required | Title of the album |
| artist | String | Required | Artist of the album |
| imageUrl | String | Required | URL to the album cover image |
| releaseYear | Number | Required | Release year of the album |
| songs | ObjectId[] (ref: Song) | Optional (array of Song IDs) | List of songs in the album |
| createdAt | Date | Auto-generated (timestamp) | Album creation time |
| updatedAt | Date | Auto-generated (timestamp) | Album update time |

**5.5 Entity Relationship (ER)Diagram:**

An entity-relationship (ER) diagram is a specialized graphic that illustrates the relationships between entities. ER diagrams often use symbols to represent different types of information.

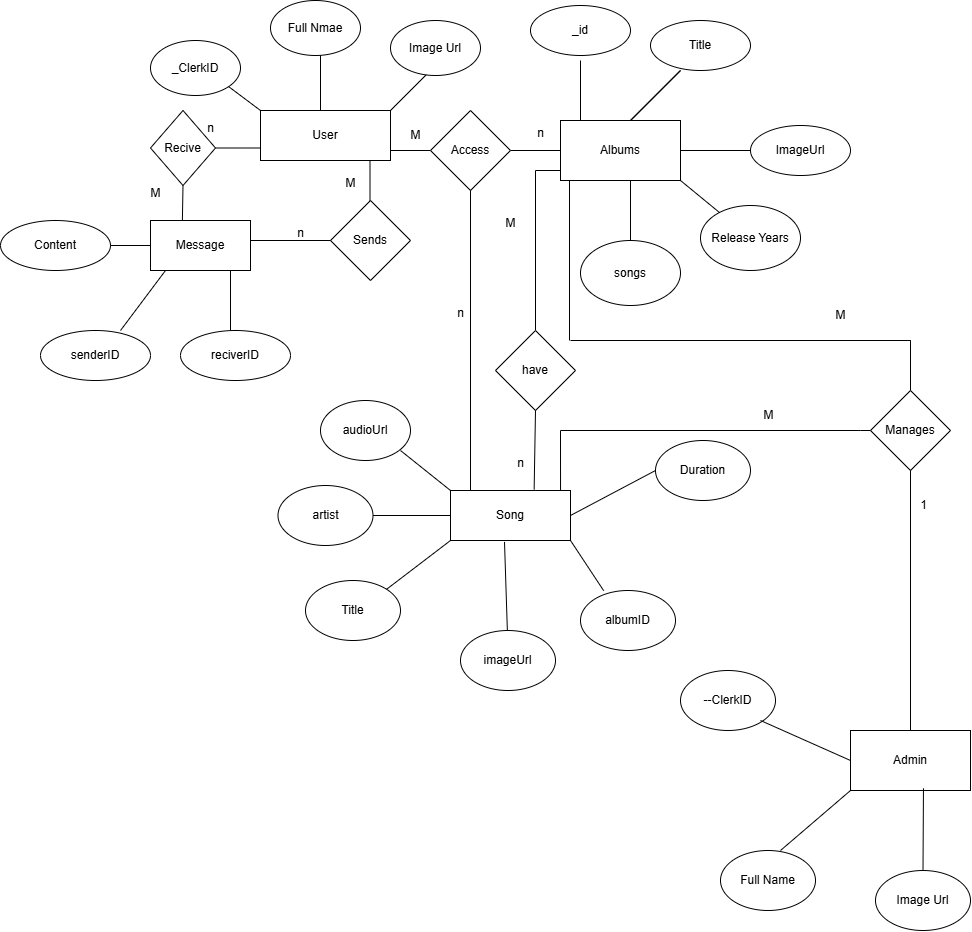
**5.6 Symbols: **

* **Entity (Rectangle):** Represents a real-world object or concept (like User or Song).
* **Weak Entity (Double Rectangle):** An entity that depends on another for identification.
* **Associative Entity (Diamond in Rectangle):** Connects two or more entities in a relationship.
* **Relationship (Diamond):** Shows a relationship between entities
* **Identifying Relationship (Double Diamond):** A relationship that helps identify a weak entity
* **Optional Relationship (Dashed Line):** A relationship that may or may not exist.
* **Mandatory Relationship (Solid Line):** A relationship that must exist.
* **Derived Attribute (Dashed Oval):** An attribute that can be derived from other attributes.
* **Key Attribute (Underlined Oval):** An attribute that uniquely identifies an entity.
* **Attribute (Oval):** A property or characteristic of an entity (e.g., name, age).
* **Partial Participation (Single Line):** Only some instances of an entity participate in the relationship.
* **Total Participation (Double Line):** All instances of an entity must participate in the relationship.

**5.7 Relationship  exist between entities:**

* **Binary Relationship or Cardinality**
* Binary Relationship means relation between two Entities. This is further divided into four types.
* **One to One:** This type of relationship is rarely seen in real world.
* **One to Many:** It reflects business rule that one entity is associated with many numbers of same entity. The example for this relation might sound a little weird, but this means that one student can enrol to many courses, but one course will have one Student.
* **Many to One:** It reflects business rule that many entities can be associated with just one entity. For example, Student enrols for only one Course, but a Course can have many Students.
* **Many to Many:** It reflects business rule that many entities can be associated with many entities.

**5.8 Entity Relationship diagram of Banger:**



**Chapter – 6  
DETAILED DESIGN**

**6.1 Introduction:**

* In details design we decide the internal logic for the module.
* Detailed design is an extensor of system design.

**Structured English Admin Login Form:**

* IF (click on login) then

Check the login and password

ELSE Message " Yor Not Admin"

END IF

  END

**User Login Form:**

Customer Login Flow (with Clerk Google OAuth):

BEGIN

IF (user clicks "Continue with Google") THEN

  Clerk handles authentication using Google OAuth

    IF (authentication is successful) THEN

        IF (user record already exists in the database) THE

            Redirect to Home Page

        ELSE

            Create new user record in the database using Clerk profile data

            Redirect to Home page

        END IF

    ELSE

        Display the message: "Google login failed. Please try again."

    END IF

END

**Chapter – 7  
SYTEM TESTING**

**7.1 Testing:**

Introduction:

* Testing is the process of detecting the errors. Testing performs a very special role for quantity assurance and for reliability of software. The results of testing are used later on during maintenance also.

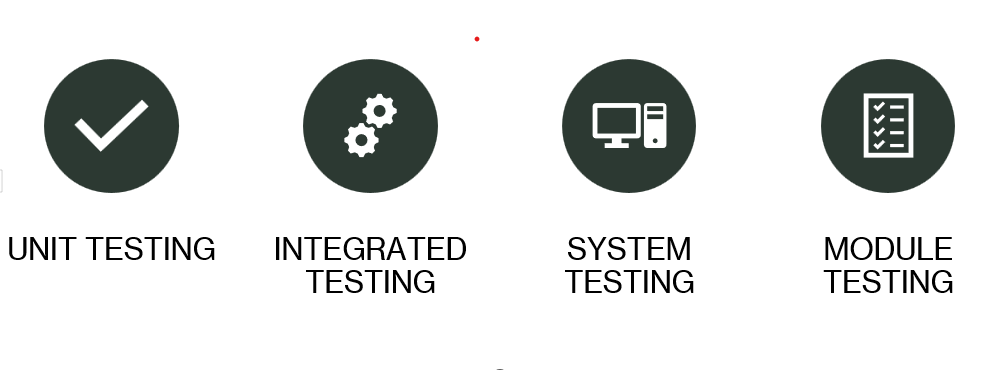
**Psychology of Testing:**

The aim of testing is often to demonstrate that a program works by showing that it has no errors. The basic purpose of testing phase is to detect the errors that may be present in the program.

**Testing of Obejectives:**

The main objectives of testing are to uncover errors, systematically and with minimum effort and time. Starting formally, we can say, testing is a process of executing a program with the intent of finding errors.

**Level Of Testing:**

**7.2 Testing Levels in Brief:**

* **Unit Testing**
* Testing of an individual unit is termed as [Unit Testing](https://www.softwaretestinghelp.com/unit-testing/). It is typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code.
* **Integrated Testing**
* Testing of all integrated modules to verify the combined functionality after integration is termed as[Integration Testing](https://www.softwaretestinghelp.com/what-is-integration-testing/).
* **System Testing**
* Under [System Testing](https://www.softwaretestinghelp.com/system-testing/), the entire system is tested as per the requirements. It is a Black-box type Testing that is based on overall requirement specifications and covers all the combined parts of a system.
* **Module Testing**
* Module testing is a type of software testing where individual model of the software are tested.

**7.3 Test Case for Admin & User  Login:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Serial No. | Condition | To be Tested | Test Data | Expected Result | Remarks |
| 1 | If 'Continue with Google' is clicked | Auth popup opens and handles login | Click login button | Google login popup appears | SUCCESSFUL |
| 2 | User cancels Google login | Auth canceled | Click 'Cancel' on Google | Stay on login page with no session created | SUCCESSFUL |
| 3 | User logs in with non-admin Gmail | Validate redirection for normal users | user@gmail.com | Redirected to home page (not admin dashboard) | SUCCESSFUL |
| 4 | User logs in with admin Gmail | Perform admin access validation | admin@gmail.com | Redirected to admin dashboard | SUCCESSFUL |
| 5 | Google login fails | OAuth failure handling issue | - | Display: 'Google login failed. Please try again.' | SUCCESSFUL |
| 6 | User already logged in | Clerk session auto-redirect | Revisit app with session | Auto-redirected to the appropriate dashboard (user or admin) | SUCCESSFUL |
| 7 | First-time login | New user creation in DB | Sign in with new Gmail | User record created, redirected to correct dashboard | SUCCESSFUL |

**7.4 Test Case of User Functionality:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Serial No. | Condition | To be Tested | Test Data | Expected Output | Remarks |
| 1 | If no account is selected during Google login | Login with Google | Cancel Google popup | User stays on login page with no session created | SUCCESSFUL |
| 2 | If a valid user logs in | Login with Google | Valid Gmail (non-admin) | User redirected to user dashboard | SUCCESSFUL |
| 3 | If no songs are available | Browse songs | - | Display: 'No songs available' | SUCCESSFUL |
| 4 | If songs are available | Listen to a song | - | Song plays | SUCCESSFUL |
| 5 | If albums are available | View album and songs | Click on album | Opens album with songs | SUCCESSFUL |
| 6 | If album has no songs | View empty album | - | Display: 'No songs in this album' | SUCCESSFUL |
| 7 | If another user is online and listening | Show presence status | - | User B sees 'User A is listening to song X' | SUCCESSFUL |
| 8 | If user is offline | Presence status | No activity | User B sees 'User A is offline' | SUCCESSFUL |
| 9 | Chat message sent to online user | Send chat message | Message to online user | Message delivered and seen | SUCCESSFUL |
| 10 | Chat message sent to offline user | Store and sync chat | Message to offline user | Message stored and shown when user comes online | SUCCESSFUL |
| 11 | User logs out and sends another message | OAuth expired | After logout | Show error: 'User not authenticated' | SUCCESSFUL |
| 12 | User logs out | Click logout button | - | Session ends and redirected to login | SUCCESSFUL |
| 13 | Someone listens to a song | Song personalization | Another song is playing | Show suggested songs based on song being played | SUCCESSFUL |
| 14 | Network drops while in session | Network disconnection | Disable internet | Display message: 'Disconnected' and retry logic | SUCCESSFUL |
| 15 | Real-time update of user status | Online/offline sync | Login/Logout of other users | UI updates user list accordingly | SUCCESSFUL |

**7.5 Table : 1 Access Control, Album Management, and Song Management**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Serial No. | Module | Condition / Feature | To Be Tested | Test Data / Action | Expected Result | Remarks |
| 1 | Admin Access Control | Admin login is fine | Check if admin access is correctly granted | Login with admin@gmail.com | Redirected to admin dashboard | SUCCESSFUL |
| 2 | Admin Access Control | Normal user logs in | Check access restriction for normal users | Login with user@gmail.com | Redirected to user homepage, denied admin access | SUCCESSFUL |
| 3 | Album Management | Create Album | Add new album with valid inputs | Title, Artist, Image URL, Year | Album saved and appears in album list | SUCCESSFUL |
| 4 | Album Management | Create Album (missing fields) | Add album with missing title | No title, other fields filled | Alert: 'Please fill in all required fields' | SUCCESSFUL |
| 5 | Album Management | Edit Album | Change title of album | Change title from 'Old' to 'New' | Album details updated successfully | SUCCESSFUL |
| 6 | Album Management | Delete Album | Remove an album | Click delete on album 'XYZ' | Album removed and songs reassigned or deleted | SUCCESSFUL |
| 7 | Song Management | View/Search Songs | Search and filter songs | Search 'Ajay' | Relevant songs shown | SUCCESSFUL |
| 8 | Song Management | Add Song | Upload song with metadata | Title, Artist, Album, Audio URL | Song added to song list | SUCCESSFUL |
| 9 | Song Management | Edit Song | Edit existing song | Change artist name | Song updated successfully | SUCCESSFUL |
| 10 | Song Management | Delete Song | Remove a song | Delete 'Tum Hi Ho' | Song deleted from database | SUCCESSFUL |
| 11 | Song Management | View/Search Songs | Search and filter by album or artist | Filter by 'Ajit Singh' | Relevant songs displayed | SUCCESSFUL |

**7.6 Table 2: User Management, Message Moderation, Dashboard, and Analytics**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Serial No. | Module | Condition / Feature | To Be Tested | Test Data / Action | Expected Result | Remarks |
| 12 | User Management | View Users | List all users | Open 'Users' section | Table of users shown | SUCCESSFUL |
| 13 | User Management | Edit User | Update profile name/image | Edit user <John> + name to 'Johnny' | Changes reflected successfully | SUCCESSFUL |
| 14 | User Management | Delete User | Remove user from system | Delete user with ID: 123 | User deleted | SUCCESSFUL |
| 15 | Message Moderation | View Messages | View user messages | Open chat moderation panel | All messages listed with sender info | SUCCESSFUL |
| 16 | Message Moderation | Delete Messages | Remove inappropriate content | Click delete on spam message | Message removed | SUCCESSFUL |
| 17 | Dashboard & Stats | Statistics | Display user/song/album/message count | Navigate to dashboard | Stats displayed (e.g. 120 users, 65 albums) | SUCCESSFUL |
| 18 | Dashboard Monitoring | Monitor Activity | Check user uploads and recent actions | View recent logs | Admins see recent user activities | SUCCESSFUL |

**Chapter – 8  
CODING**

**8.1 Front end:**

import { Route, Routes } from "react-router-dom";

import HomePage from "./pages/home/HomePage";

import AuthCallbackPage from "./pages/auth-callback/AuthCallbackPage";

import { AuthenticateWithRedirectCallback } from "@clerk/clerk-react";

import MainLayout from "./layout/MainLayout";

import ChatPage from "./pages/chat/ChatPage";

import AlbumPage from "./pages/album/AlbumPage";

import AdminPage from "./pages/admin/AdminPage";

import { Toaster } from "react-hot-toast";

import NotFoundPage from "./pages/404/NotFoundPage";

function App() {

    return (

        <>

            <Routes>

                <Route

                    path='/sso-callback'

                    element={<AuthenticateWithRedirectCallback signUpForceRedirectUrl={"/auth-callback"} />}

                />

              <Route path='/auth-callback' element={<AuthCallbackPage />} />

                <Route path='/admin' element={<AdminPage />} />

                <Route element={<MainLayout />}>

                    <Route path='/' element={<HomePage />} />

                    <Route path='/chat' element={<ChatPage />} />

                    <Route path='/albums/:albumId' element={<AlbumPage />} />

                    <Route path='\*' element={<NotFoundPage />} />

                </Route>

            <Toaster

  </>

    );

}

export default App;

import { StrictMode } from "react";

import { createRoot } from "react-dom/client";

import "./index.css";

import App from "./App.tsx";

import { ClerkProvider } from "@clerk/clerk-react";

import { BrowserRouter } from "react-router-dom";

import AuthProvider from "./providers/AuthProvider.tsx";

const PUBLISHABLE\_KEY = import.meta.env.VITE\_CLERK\_PUBLISHABLE\_KEY;

if (!PUBLISHABLE\_KEY) {

    throw new Error("Missing Publishable Key");

}

createRoot(document.getElementById("root")!).render(

    <StrictMode>

        <ClerkProvider publishableKey={PUBLISHABLE\_KEY} afterSignOutUrl='/'>

            <AuthProvider>

                <BrowserRouter>

                    <App />

                </BrowserRouter>

            </AuthProvider>

        </ClerkProvider>

    </StrictMode>

);

**8.2 Backend:**

import { Song } from "../models/song.model.js";

import { Album } from "../models/album.model.js";

import cloudinary from "../lib/cloudinary.js";

// helper function for cloudinary uploads

const uploadToCloudinary = async (file) => {

    try {

        const result = await cloudinary.uploader.upload(file.tempFilePath, {

            resource\_type: "auto",

        });

        return result.secure\_url;

    } catch (error) {

        console.log("Error in uploadToCloudinary", error);

        throw new Error("Error uploading to cloudinary");

    }

};

export const createSong = async (req, res, next) => {

    try {

        if (!req.files || !req.files.audioFile || !req.files.imageFile) {

            return res.status(400).json({ message: "Please upload all files" });

        }

        const { title, artist, albumId, duration } = req.body;

        const audioFile = req.files.audioFile;

        const imageFile = req.files.imageFile;

        const audioUrl = await uploadToCloudinary(audioFile);

        const imageUrl = await uploadToCloudinary(imageFile);

        const song = new Song({

            title,

            artist,

            audioUrl,

            imageUrl,

            duration,

            albumId: albumId || null,

        });

        await song.save();

        // if song belongs to an album, update the album's songs array

        if (albumId) {

            await Album.findByIdAndUpdate(albumId, {

                $push: { songs: song.\_id },

            });

        }

        res.status(201).json(song);

    } catch (error) {

        console.log("Error in createSong", error);

        next(error);

    }

};

export const deleteSong = async (req, res, next) => {

    try {

        const { id } = req.params;

        const song = await Song.findById(id);

        // if song belongs to an album, update the album's songs array

        if (song.albumId) {

            await Album.findByIdAndUpdate(song.albumId, {

                $pull: { songs: song.\_id },

            });

        }

        await Song.findByIdAndDelete(id);

        res.status(200).json({ message: "Song deleted successfully" });

    } catch (error) {

        console.log("Error in deleteSong", error);

        next(error);

    }

};

export const createAlbum = async (req, res, next) => {

    try {

        const { title, artist, releaseYear } = req.body;

        const { imageFile } = req.files;

        const imageUrl = await uploadToCloudinary(imageFile);

        const album = new Album({

            title,

            artist,

            imageUrl,

            releaseYear,

        });

        await album.save();

        res.status(201).json(album);

    } catch (error) {

        console.log("Error in createAlbum", error);

        next(error);

    }

};

export const deleteAlbum = async (req, res, next) => {

    try {

        const { id } = req.params;

        await Song.deleteMany({ albumId: id });

        await Album.findByIdAndDelete(id);

        res.status(200).json({ message: "Album deleted successfully" });

    } catch (error) {

        console.log("Error in deleteAlbum", error);

        next(error);

    }

};

export const checkAdmin = async (req, res, next) => {

    res.status(200).json({ admin: true });

};

// backend/src/controller/admin.controller.js

export const updateSong = async (req, res, next) => {

    try {

        const { id } = req.params;

        const { title, artist, duration, albumId } = req.body;

        const { audioFile, imageFile } = req.files;

        const song = await Song.findById(id);

        if (!song) {

            return res.status(404).json({ message: "Song not found" });

        }

        if (audioFile) {

            const audioUrl = await uploadToCloudinary(audioFile);

            song.audioUrl = audioUrl;

        }

        if (imageFile) {

            const imageUrl = await uploadToCloudinary(imageFile);

            song.imageUrl = imageUrl;

        }

        song.title = title;

        song.artist = artist;

        song.duration = duration;

        song.albumId = albumId || null;

        await song.save();

        res.status(200).json(song);

    } catch (error) {

        console.log("Error in updateSong", error);

        next(error);

    }

};

export const updateAlbum = async (req, res, next) => {

    try {

        const { id } = req.params;

        const { title, artist, releaseYear } = req.body;

        const { imageFile } = req.files;

        const album = await Album.findById(id);

        if (!album) {

            return res.status(404).json({ message: "Album not found" });

        }

        if (imageFile) {

            const imageUrl = await uploadToCloudinary(imageFile);

            album.imageUrl = imageUrl;

        }

        album.title = title;

        album.artist = artist;

        album.releaseYear = releaseYear;

        await album.save();

        res.status(200).json(album);

    } catch (error) {

        console.log("Error in updateAlbum", error);

        next(error);

    }

};

import { clerkClient } from "@clerk/express";

export const protectRoute = async (req, res, next) => {

    if (!req.auth.userId) {

        return res.status(401).json({ message: "Unauthorized - you must be logged in" });

    }

    next();

};

**Chapter – 9  
SNAPSHOTS**

**9.1 Home Page Banger:**

**A screenshot of a music playlist

AI-generated content may be incorrect.**

**9.2 Chat Page:** A screenshot of a video chat

AI-generated content may be incorrect.

**9.3 Admin Dashboard:**

A screenshot of a music album

AI-generated content may be incorrect.

**9.4 Add Song:**

A screenshot of a music player

AI-generated content may be incorrect.

**9.5 Edit Song:**

A screenshot of a music album

AI-generated content may be incorrect.

**9.6 Album Manage:**

A screenshot of a computer

AI-generated content may be incorrect.

**9.7 Add Album:**

A screenshot of a music album

AI-generated content may be incorrect.

**9.8 Edit Album:**

A screenshot of a music album

AI-generated content may be incorrect.

**Chapter – 10  
LIMITATION**

* **Internet Dependency**
* Requires a stable internet connection for streaming, with no offline mode available yet.
* **Device Compatibility**
* Currently only available as a web application, with mobile and other platform versions still under development.
* **Limited Social Features**
* While real-time chat is available, advanced social features like collaborative playlists or group listening are not yet implemented.
* **Storage Limitations**
* The amount of space available for user-created playlists or custom data might be limited, depending on the storage solution used.
* **Quality Control**
* Variations in the audio quality of streaming could occur due to network conditions or the availability of high-quality sources.

**Chapter – 11**

**FUTURE ENHANCEMENT**

* **Mobile Application Development** – Expand Banger to portable devices like smartphones and tablets for seamless music streaming on the go.
* **AI-Based Music Recommendations** – Implement AI-driven personalized song and album recommendations based on user preferences and listening history.
* **AI-Based Music Recommendations** – Implement AI-driven personalized song and album recommendations based on user preferences and listening history.
* **Live Streaming & Artist Interaction** – Introduce live sessions where artists can interact with users and stream exclusive content.
* **Voice Commands & Smart Assistant Integration** – Allow users to control playback and search for songs using voice commands.

**Chapter – 12**

**CONCLUSION**

The Banger project successfully delivers a dynamic music streaming experience using the MERN stack.  
 It supports user login via Google, real-time chat, and live song status updates.  
 Admins can efficiently manage songs, albums, and users through a dedicated dashboard.  
 With future enhancements like offline support and premium features, the platform can become even more robust and user-friendly.

**Chapter – 13**

**BIBLIOGRAPHY**

**References and Bibliography:**

* Mosh Hamedani. *"MERN Stack Front-To-Back: Full Stack React, Redux & Node.js.*" Udemy, 2020.
* Traversy Media. *"MERN Stack Tutorial | Full Stack React, Node, Express & MongoDB."* YouTube, 2019.

"MERN Stack Documentation." *MongoDB, Express, React, Node.js.* MongoDB, 2022

**Web References**

MongoDB **-** https://www.mongodb.com/resources/languages/mern-stack

 Vite- <https://vite.dev/>

React-vite :https://react.dev/learn

Banger Site Link: https://banger-litee.onrender.com