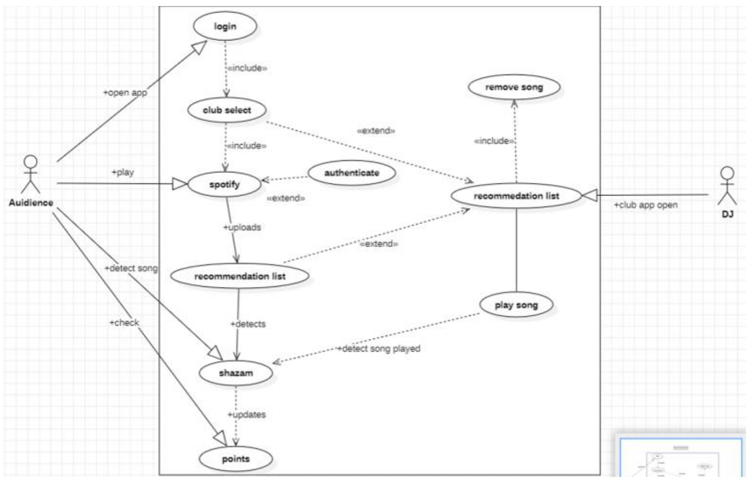
**SRS** stands for **Software Requirements Specification**. It is a document that describes the functionality, features, and behavior of a software system that is to be developed. It serves as a blueprint for both developers and stakeholders, outlining the requirements for the system from both functional and non-functional perspectives. An SRS document helps in ensuring that all stakeholders have a clear and shared understanding of the software's functionality and requirements before development begins.

The SRS document typically includes:

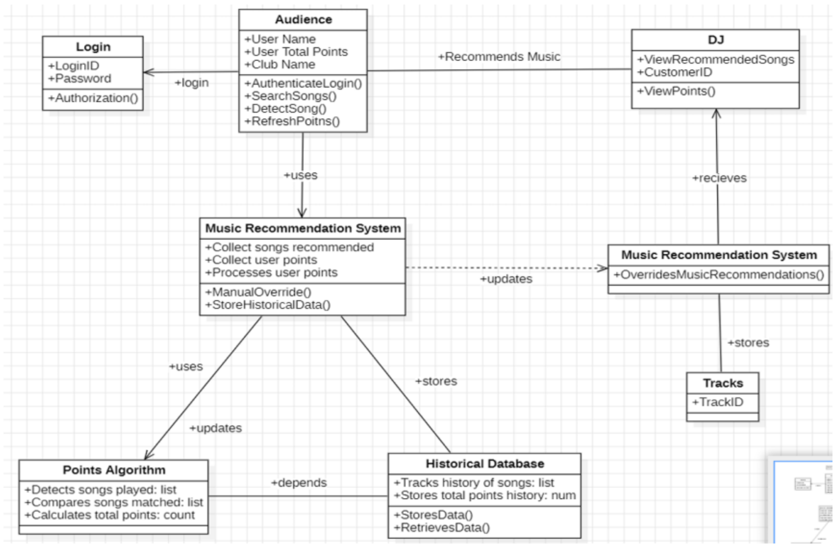
1. **Introduction**
2. **System Features**
3. **External Interfaces**
4. **User Requirements**
5. **System Architecture**
6. **Non-functional Requirements**
7. **Constraints and Assumptions**

**UML (Unified Modeling Language)**

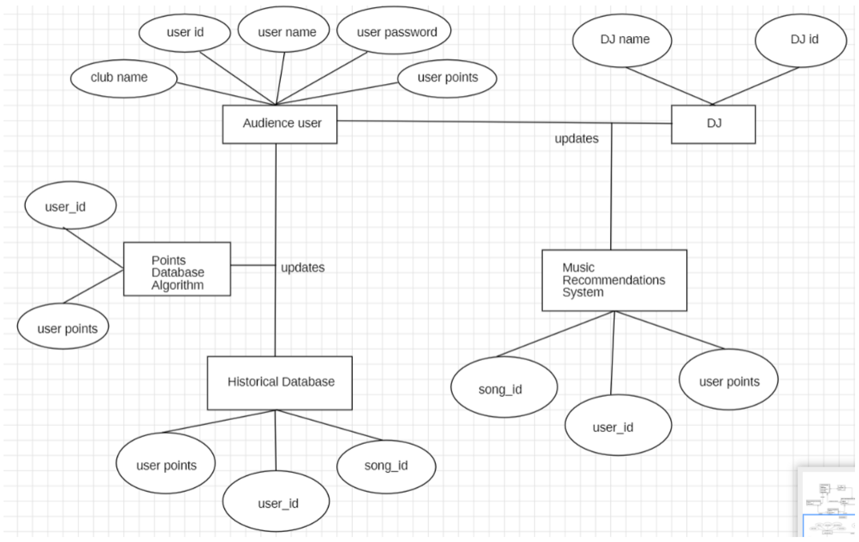
1. **Use Case Diagram**: Represents the interactions between users (actors) and the system. It shows the system's functionality from an external user's perspective, identifying the system's goals and boundaries.



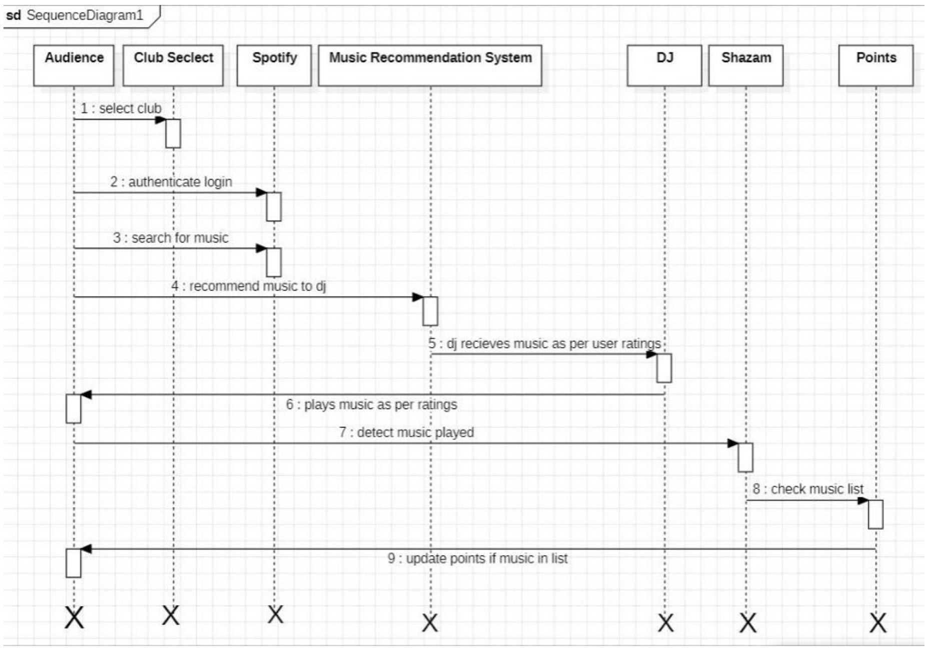
2. **Class Diagram**: Depicts the structure of the system by showing its classes, their attributes, methods, and relationships. It helps visualize how objects are created and interact within the system.



3. **ER (Entity-Relationship) Diagram**: Shows the relationships between data entities in a database. It helps in designing and visualizing the database structure, including entities, relationships, and attributes.

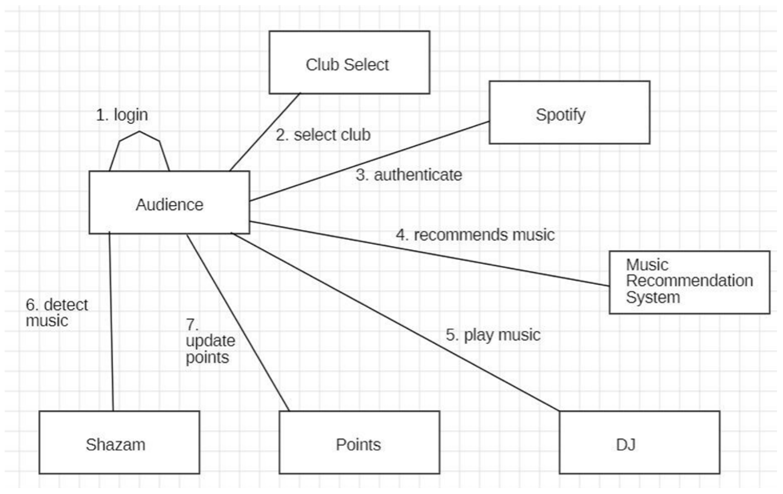


4. **Sequence Diagram**: Illustrates how objects interact with each other in a sequential order over time. It shows the flow of messages and method calls between objects in a specific scenario.



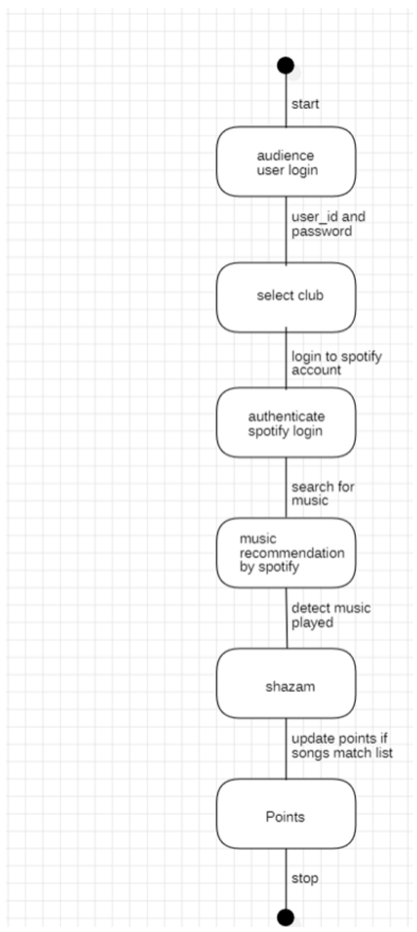
5. **Collaboration Diagram**: Similar to sequence diagrams but focuses more on the relationships and structure of objects in the system. It shows how objects collaborate to achieve a specific behavior.

Is last diagram me khudse arrows bana dena pencil / pen

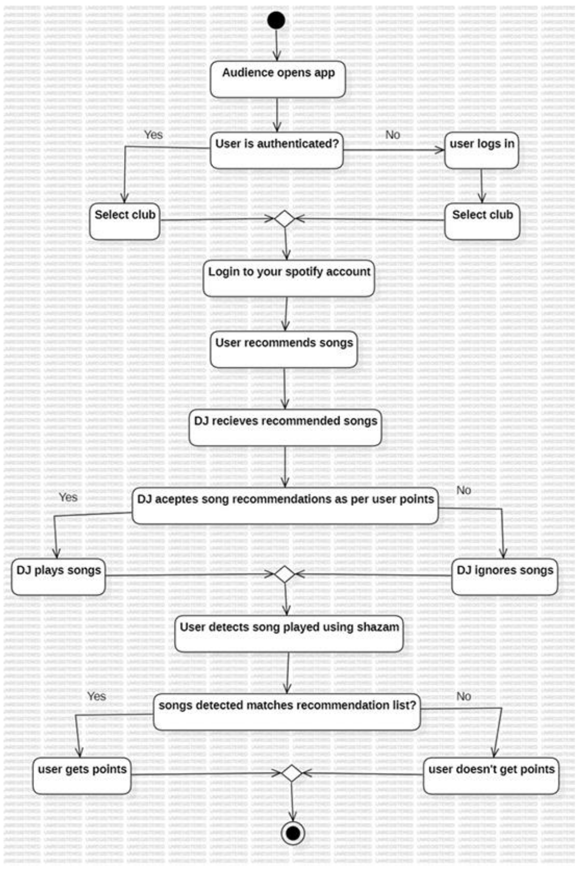


6. **Statechart (State Machine) Diagram**: Represents the different states an object or system can be in and the transitions between those states. It's useful for modeling the life cycle of an object or system component.

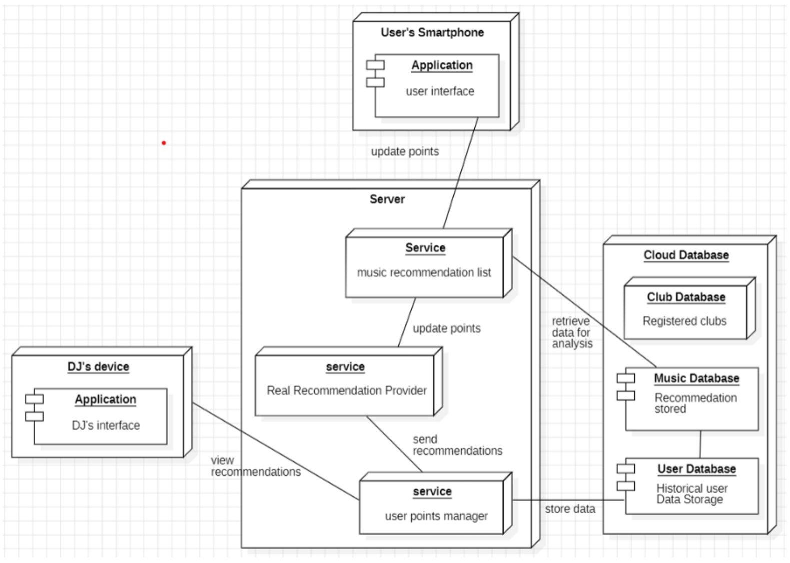
State diagram: arrows bana dena un connecting lines pe



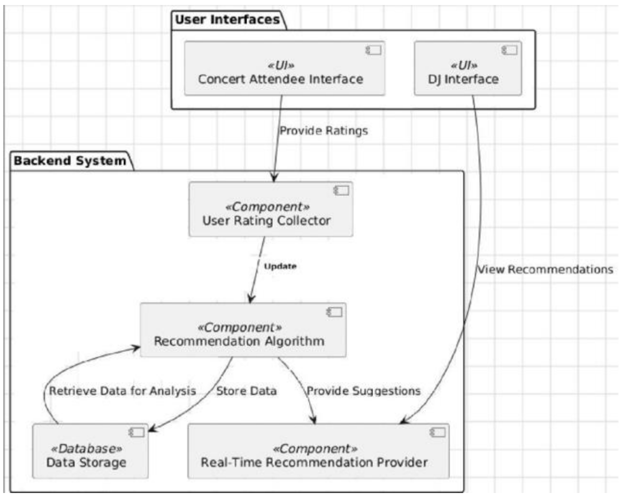
7. **Activity Diagram**: Describes the dynamic aspects of the system. It shows the flow of control or data between activities, similar to a flowchart, and is used to represent workflows or business processes.


Audience opens app
Yes
User is authenticated?
No
user logs in
Select club
Select club
Login to your spotify account
User recommends songs
DJ recieves recommended songs
Yes
DJ aceptes song recommendations as per user points
No
DJ plays songs
DJ ignores songs
User detects song played using shazam
Yes
songs detected matches recommendation list?
No
user gets points
user doesn't get points

8. **Deployment Diagram**: Focuses on the physical aspects of the system, showing the hardware components (nodes) and how software components (artifacts) are deployed on them.

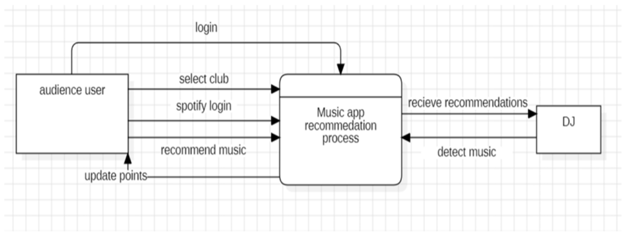


9. **Component Diagram**: Depicts the components of the system (e.g., software modules or subsystems) and their interactions. It highlights the system's structure at a higher level, helping in understanding how different parts work together.

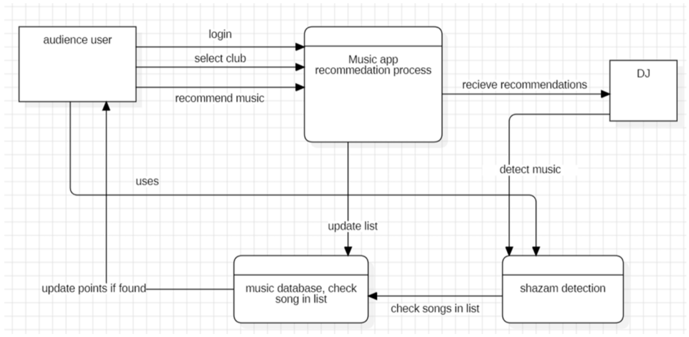

User Interfaces
«UI» Concert Attendee Interface
«UI» DJ Interface
Provide Ratings
Backend System
«Component» User Rating Collector
Update
View Recommendations
«Component» Recommendation Algorithm
Retrieve Data for Analysis
Store Data
Provide Suggestions
«Database» Data Storage
«Component» Real-Time Recommendation Provider

10. **Data Flow Diagram**: Represents the flow of data through a system, focusing on the processes that transform the data. It shows how inputs are converted into outputs and helps identify data movement and storage.

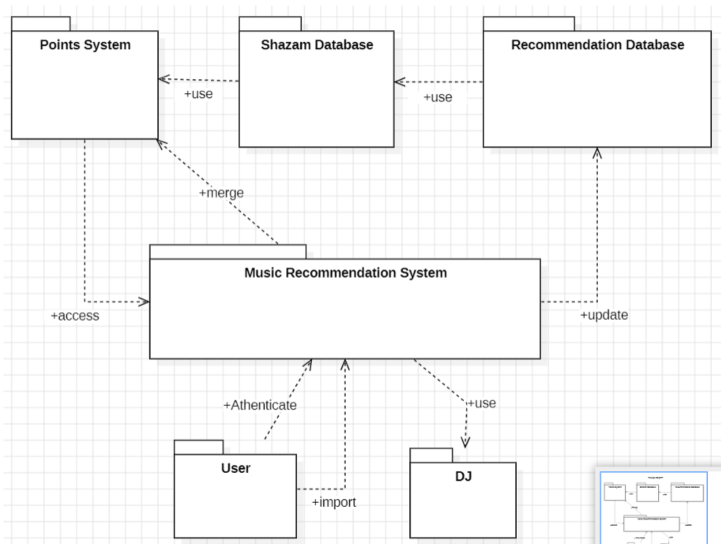
O level dfd



1 level dfd



11. **Package Diagram**: Shows how the system is divided into packages (groups of related classes or components). It helps visualize the modular structure of a system and the dependencies between different packages.



- **White-box testing** is a method of testing where the tester has access to the internal workings of the system, including the code, algorithms, and logic. The tester tests the internal functions and the structure of the application.  The focus is on the code and logic of the software, ensuring that all paths, branches, loops, and statements are executed correctly. Helps identify hidden errors within the code. Requires deep knowledge of the code, making it time-consuming and complex.
- **Black-box testing** is a method of testing where the tester does not have access to the internal workings or code of the system. Instead, the tester focuses on testing the software's functionality based on the requirements and specifications. The focus is on the output generated by the software for given inputs, ensuring the software behaves as expected under various conditions. No knowledge of the internal code is required, making it easier to perform from an end-user perspective. The tester may need to create many test cases to cover all possible input combinations.