Lab 4: Architecture Kata Report

Team Size: **2** students

student 1: **FINICHIU** Eduard - Adelin

student 2: **JERCAU** Hadasa-Stefana

FILS **1241** E**B**

Selected Scenario: **Music App for Shared Playlists and Synchronized Listening**

1. High-Level Architecture

Design Pattern: Event-Driven Microservices

Rationale: We chose this architecture because the application has two distinct behaviors with conflicting needs:

1. Playlist Management: Standard CRUD operations (Create, Read, Update, Delete) which require data consistency
2. Live Syncing: Real-time, high-frequency data (updates every second) which requires low latency.

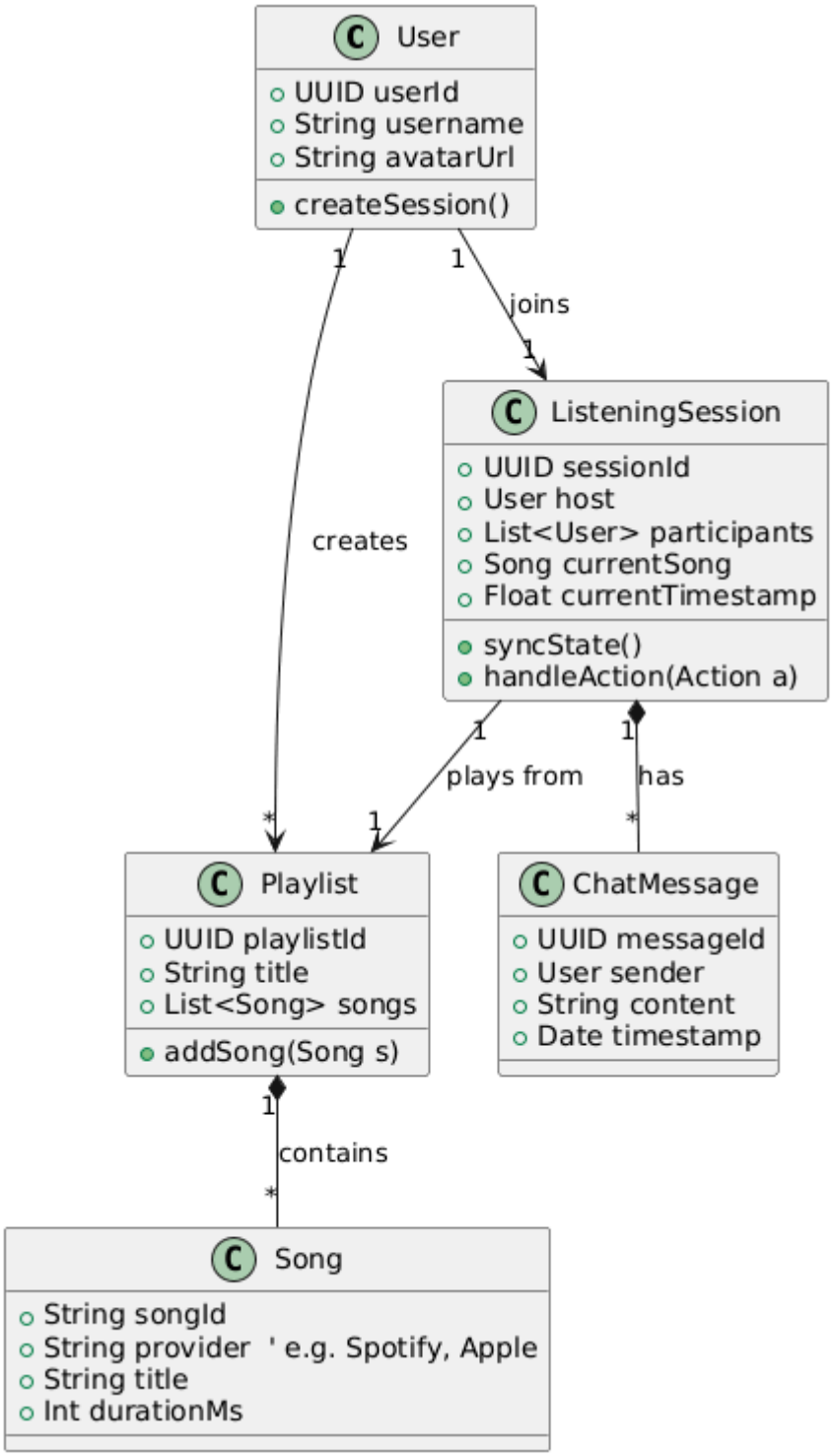
Separating these into different services allows us to scale the "Socket Service" (for live listening) independently from the "Playlist Service" (for database storage).

System Components:

- Mobile Client: Uses local state to manage the UI and SDKs (Spotify/Apple Music) to play audio.
- API Gateway: Routes HTTP requests and handles SSL termination.
- Auth Service: Manages user identity and issues JWTs.
- Playlist Service: Manages the permanent database of songs and user libraries.
- Socket Service (The "Sync Engine"): A dedicated service using WebSockets to keep all users in a "room" on the same timestamp.
- Redis (Cache/PubSub): Used for ephemeral state (e.g., "User A is at 0:45", "User B is at 0:46").

2. Class Diagram

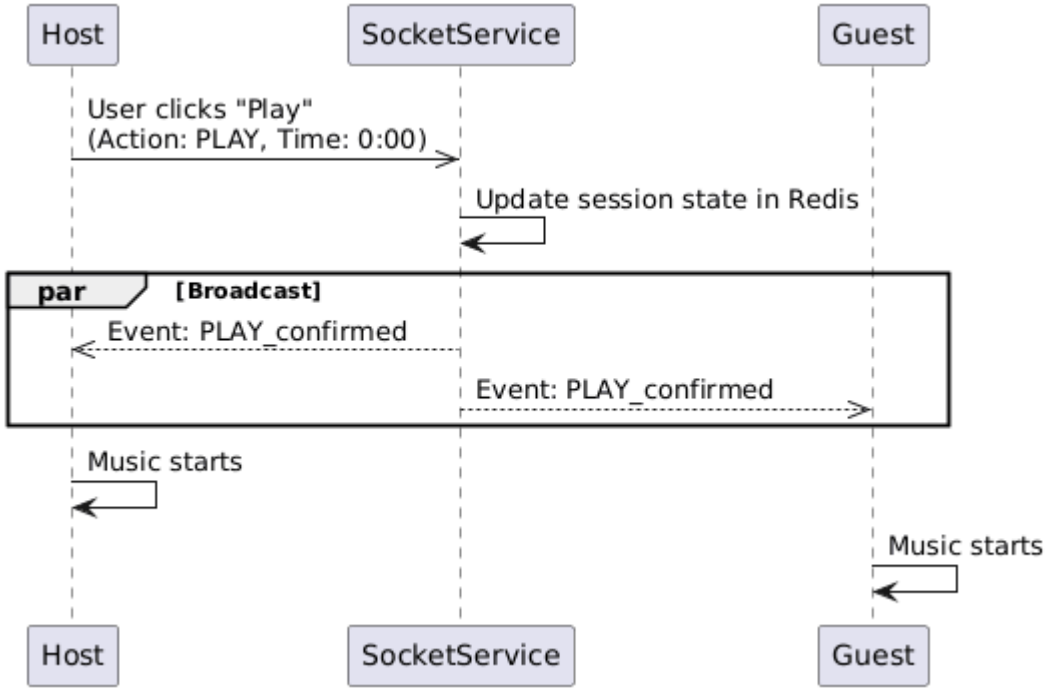
This diagram highlights the relationship between the permanent data (Users/Playlists) and the temporary session data (ListeningSession).



3. Sequence Diagrams

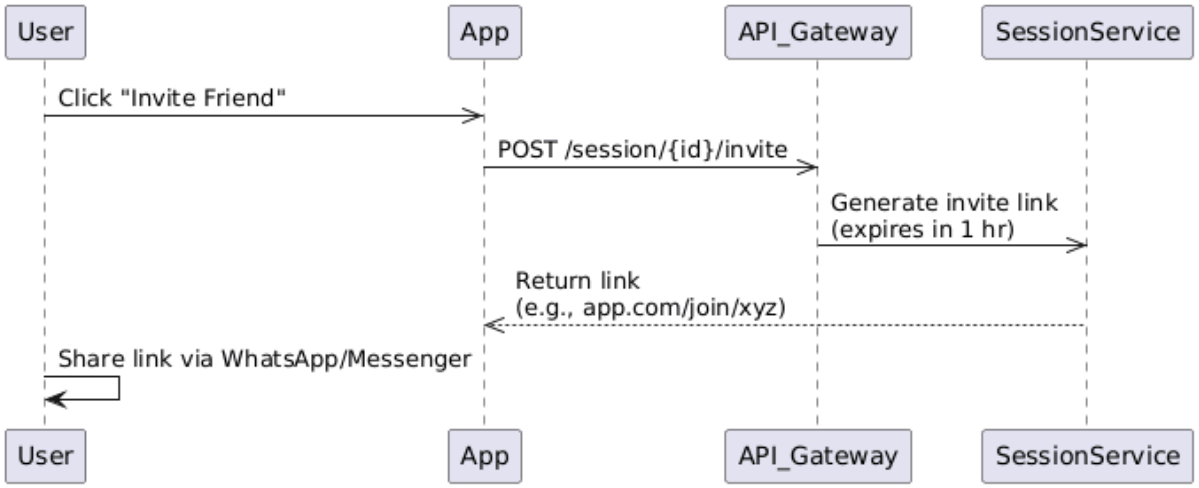
Use Case A: Syncing Logic (The "Anti-Lag" Strategy)

When a user presses "Play", we don't just play the music. We send a signal to the server, which then tells everyone (including the person who pressed play) to start. This ensures total synchronization.



Use Case B: Inviting a Friend

This shows how we handle the social aspect and security (verifying the invite code).



4. Data Security & Scalability

Data Security

1. OAuth 2.0 Integration: We will not store user passwords or credit card info. We will use OAuth to link with Spotify/Apple Music accounts. This delegates the heavy security lifting to the providers.
2. Secure WebSockets (WSS): All real-time traffic is encrypted. If someone sniffs the Wi-Fi traffic, they cannot see the chat messages or the invite tokens.
3. Role-Based Access Control: Only the Host of the session is authorized to Kick User or Change Mode (e.g., switching from "Dictator Mode" where only the host picks songs, to "Democracy Mode" where anyone can pick).

Scalability Strategy

1. Redis for State: We use Redis to store the current playback state. It is much faster than a SQL database. If 10,000 users are syncing, we need sub-millisecond read/write speeds that SQL cannot provide.
2. Horizontally Scaling Sockets: We can spin up multiple "Socket Nodes". If Node A fails, the users are automatically reconnected to Node B. We use a Load Balancer with sticky sessions to manage this traffic.

5. Team Presentation

For our solution, we chose the Music App scenario. We split the architecture into two main parts: the REST API for managing playlists and a WebSocket layer for the real-time sync. Class Diagram ensures we have a clear separation between permanent data, like Playlists, and ephemeral data, like the active ListeningSession. We decided that the ListeningSession class should not be stored in the main database, but in a cache like Redis to ensure it's fast enough. The biggest challenge was the 'latency' issue, how to make sure users hear the music at the same time. In the sequence diagram, we showed that the Client doesn't play the music immediately; it waits for the Server to confirm the timestamp. For security, we implemented OAuth 2.0 so we don't have to handle sensitive password data, and we use Secure WebSockets to encrypt the group chat.