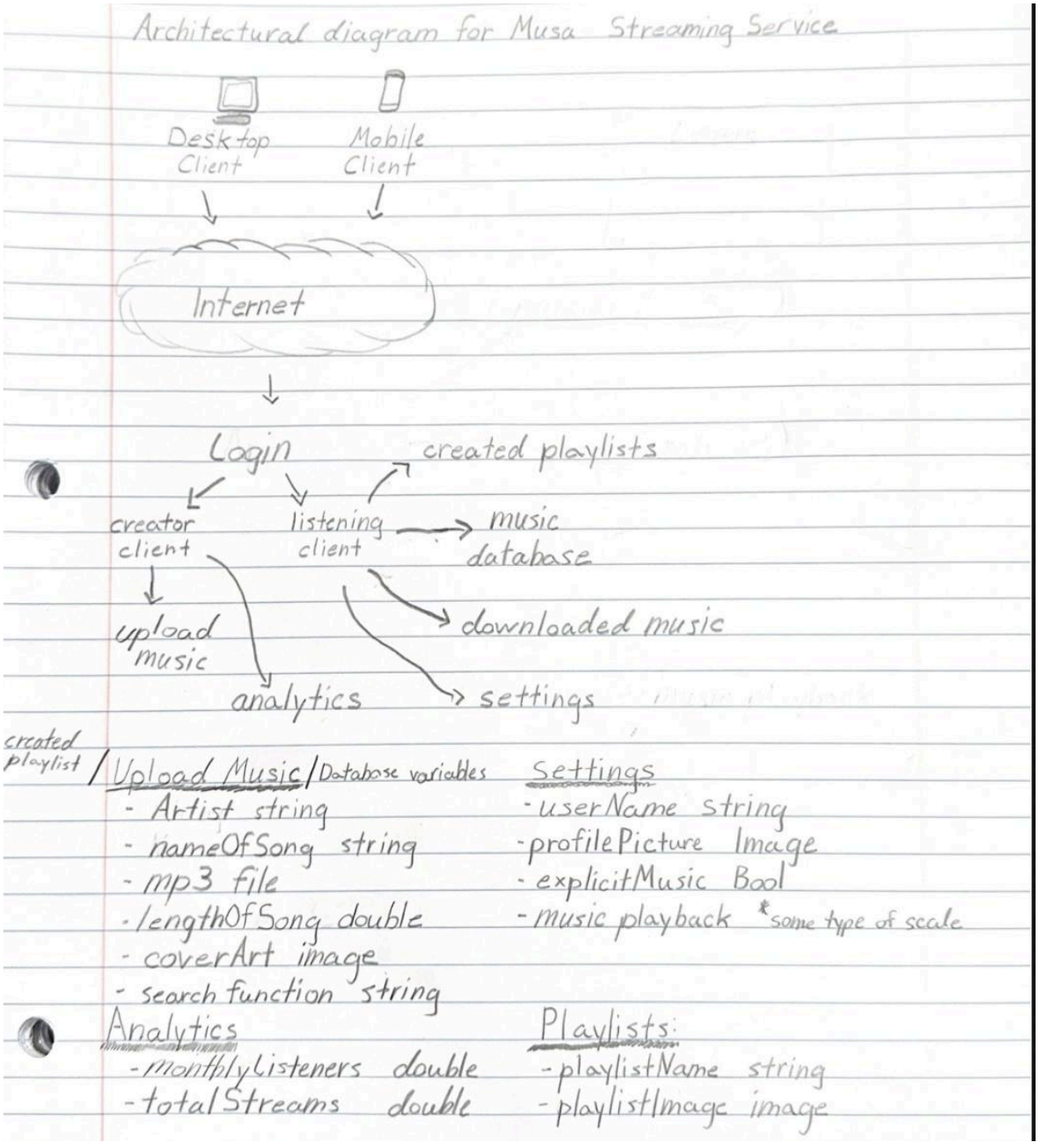
Software Design 2.0

Guillermo Lorenzo Ayala
Ali Musa

1. Software Architecture Diagram



SQL Database Management Schema

```
CREATE TABLE User (
    user_id INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE,
    password_hash VARCHAR(255) NOT NULL,
    subscription_status VARCHAR(50),
    preferences JSON -- Example of storing preferences as JSON (can be customized)
);

CREATE TABLE Artist (
    artist_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    biography TEXT,
    genre VARCHAR(255)
);

CREATE TABLE Album (
    album_id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    artist_id INT,
    release_date DATE,
    cover_image VARCHAR(255),
    FOREIGN KEY (artist_id) REFERENCES Artist(artist_id)
);

CREATE TABLE Song (
    song_id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    artist_id INT,
    album_id INT,
    duration INT, -- Duration in seconds (example)
    genre VARCHAR(255),
    release_date DATE,
    file_path VARCHAR(255),
    FOREIGN KEY (artist_id) REFERENCES Artist(artist_id),
    FOREIGN KEY (album_id) REFERENCES Album(album_id)
);

CREATE TABLE Playlist (
    playlist_id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    user_id INT,
    creation_date DATE,
```

```
    privacy_status VARCHAR(50),
    FOREIGN KEY (user_id) REFERENCES User(user_id)
);

CREATE TABLE PlaylistSong (
    playlist_song_id INT PRIMARY KEY AUTO_INCREMENT,
    playlist_id INT,
    song_id INT,
    FOREIGN KEY (playlist_id) REFERENCES Playlist(playlist_id),
    FOREIGN KEY (song_id) REFERENCES Song(song_id)
);
```

Example SQL Querying based on our Schema

```
-- Inserting a User
INSERT INTO User (username, email, password_hash, subscription_status, preferences)
VALUES ('john_doe', 'john@example.com', 'hashed_password', 'active', '{"theme": "dark",
"autoplay": true}');

-- Inserting an Artist
INSERT INTO Artist (name, biography, genre)
VALUES ('Taylor Swift', 'American singer-songwriter', 'Pop');

-- Inserting an Album
INSERT INTO Album (title, artist_id, release_date, cover_image)
VALUES ('1989', 1, '2014-10-27', 'album_covers/1989.jpg');

-- Inserting a Song
INSERT INTO Song (title, artist_id, album_id, duration, genre, release_date, file_path)
VALUES ('Shake It Off', 1, 1, 219, 'Pop', '2014-08-18', 'song_files/shake_it_off.mp3');

-- Creating a Playlist and Adding Songs
INSERT INTO Playlist (title, user_id, creation_date, privacy_status)
VALUES ('My Favorites', 1, NOW(), 'private');

-- Add songs to the playlist
INSERT INTO PlaylistSong (playlist_id, song_id)
VALUES (1, 1); -- Adding 'Shake It Off' to 'My Favorites' playlist

-- Fetching User Information
SELECT * FROM User WHERE username = 'john_doe';

-- Fetching Artist and Album Information
SELECT Artist.name AS artist_name, Album.title AS album_title
```

```
FROM Artist
INNER JOIN Album ON Artist.artist_id = Album.artist_id
WHERE Artist.name = 'Taylor Swift';

-- Fetching Songs in a Playlist
SELECT Song.title AS song_title, Artist.name AS artist_name
FROM PlaylistSong
INNER JOIN Song ON PlaylistSong.song_id = Song.song_id
INNER JOIN Artist ON Song.artist_id = Artist.artist_id
WHERE PlaylistSong.playlist_id = 1; -- Assuming playlist_id 1 is 'My Favorites'

-- Counting Total Songs per Artist
SELECT Artist.name AS artist_name, COUNT(Song.song_id) AS total_songs
FROM Artist
LEFT JOIN Song ON Artist.artist_id = Song.artist_id
GROUP BY Artist.artist_id, Artist.name;

-- Updating User's Subscription Status
UPDATE User
SET subscription_status = 'inactive'
WHERE user_id = 1; -- Assuming user_id 1 is 'john_doe'

-- Deleting a Song from a Playlist
DELETE FROM PlaylistSong
WHERE playlist_id = 1 AND song_id = 1; -- Removing 'Shake It Off' from 'My Favorites' playlist
```

2. Data Management Strategy

Database Choice:

Single Relational Database (MySQL, PostgreSQL):

Design Decision: Use a single relational database to store all application data (users, artists, albums, songs, playlists) due to simplicity and scalability for a moderate-sized application.

Data Organization: Store all related entities (users, artists, albums, songs) in normalized tables within a single database schema.

Tradeoffs: Simplified data management and transactions, but potential scalability challenges with high volume and complex queries.

Logical Data Organization:

Users: Store user information including username, email, password hash, subscription status, and preferences.

Artists: Capture artist details such as name, biography, and genre.

Albums: Store album information with a link to the corresponding artist and release details.

Songs: Hold song details like title, artist, album, duration, genre, release date, and file path.

Playlists: Manage playlists created by users with titles, creation dates, and privacy statuses.

Playlist Songs (Many-to-Many Relationship): Create associations between playlists and songs to support multiple songs per playlist.

Design Decisions and Tradeoffs:

Single Database vs. Distributed Databases:
Decision: Opted for a single relational database for simplicity.
Tradeoff: Simplified data management and transactions, but potential scalability challenges with increasing data volume.

Normalization vs. Denormalization:
Decision: Applied normalization principles to reduce redundancy and ensure data integrity.
Tradeoff: Improved data integrity and reduced redundancy but may require more complex queries for certain operations.

Relational Database vs. NoSQL Database:
Decision: We choose a relational database (e.g., MySQL, PostgreSQL) for structured data and SQL support.
Tradeoff: SQL databases provide strong consistency and ACID transactions but may be less flexible for certain types of data.

Data Partitioning and Sharding:
Decision: Not implemented data partitioning or sharding due to the current scale and simplicity of the application.
Tradeoff: Simplified data management but limited scalability in the long term.

Possible Alternatives:
Distributed Database (e.g., Cassandra, MongoDB): Offers scalability and flexibility for large-scale applications but requires more complex management and consistency considerations.

Microservices Architecture: Decompose the application into microservices, each with its own database, for more flexibility and scalability.

In summary, the chosen data management strategy revolves around a single relational database for simplicity and moderate scalability. The schema is designed to normalize data and ensure integrity, but alternative approaches could be explored for larger-scale or more complex applications. Tradeoffs exist between simplicity, scalability, and flexibility based on the current requirements and future growth expectations of the music streaming service. Adjustments and enhancements can be made as the application evolves and user demands change.