Team: Grey Orange

Members: Ryan Soeyadi <rs4163>
Joseph Rebagliati <jr4162>
Yuhao Dong <yd2626>
Madison Thantu <mgt2143>

Part 1:

State the date on which you met with your Mentor. This should be in the past, not the future.

Answer:

9/30/22

Part 2:

2a. Reconsider your answers to the part 2 questions in the preliminary proposal. Elaborate in more detail the main features your server will provide.

2b. Describe at least three different kinds of clients, including what you envision those clients would 'do' for their own clients or end-users.

Our answers to 2a and 2b are combined into the below.

1. **Service Description**
   - What our service does
     - i. Music playlist management service
   - Implementation details
     - i. Back-end:
       1. Postgres database
       2. Java server that includes both CRUD functionality and Data Analytics (see below for more details)
          - a. RESTful API used by Java server to communicate with possible clients
     - ii. Features of the service include:
       1. CRUD (**Client 1**)
          - a. creating, updating, deleting users, playlists, groups, and songs
          - b. creating analytics relation entries (see section titled "Data and Schema" for more details)

c. specific functionality is group-dependent (see section titled "Types of Clients" for more details)

2. Data Analytics (**Client 2**)
    a. aggregation and calculation functions:
        i. Most popular songs
        ii. Average song length
        iii. Least listened to songs
        iv. Most liked songs
        v. Genre with artists that are most listened to
        vi. Total number of songs played (can also be in terms of total time)
3. Data Supply (**Client 3**)
    a. Supply *raw* data that can be imported for third party developers' own purposes
        i. Music Library Supply
        ii. User profile export

2. **Type of Clients**
    i. **Client 1** (Music companies looking to launch their own isolated music platform):
        1. Upload songs
        2. Create Profiles
        3. Create playlists
        4. Join groups

Client 1 will be able to access standard CRUD functionality including the ability to insert music tracks, populate their profiles, consolidate tracks into a single playlist as well as join groups of users based on similar interests, among various other functionalities. **Client 1** groups include music companies. These groups would ping our API to develop their own frontends.

ii. **Client 2** (Data Analytic or advertisement websites calling our API to retrieve large calculated data):
    1. Getting list of popular songs
    2. Average length of the songs in a playlist
    3. Average user ages or population in different age groups

Client 2 will be able to access unique and interesting metrics via data analytics endpoints. Unlike **Client 1**, **Client 2** includes the actual data analytical or advertisement websites calling our API to retrieve a large amount of calculated data. **Client 2** is not concerned with simply retrieving raw data but data that has been compiled and analyzed. This would allow for **Client 2** to access data that they can use to target specific groups based on our response. Since this client is not a standard user base, the groups categorized as **Client 2** would likely access our data through direct API calls.

iii. **Client 3** (Other apps/third party developers sharing the same user or websites wanting to access our library)
   1. Import our song libraries
   2. Sharing user information to be used on different websites using shared profile data.

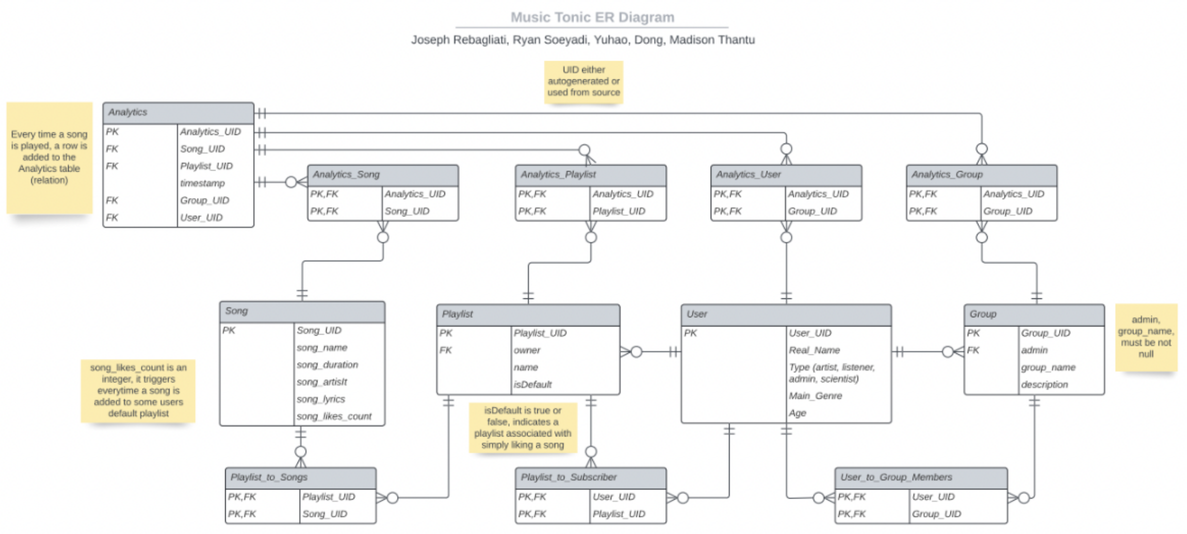
Client 3, unlike **Client 2**, would not require cleaned datasets (I.e., data that has been analyzed and computed). The data that **Client 3** receives could be used to create interesting metrics on their side, but we will not provide that service. For example, websites that require users to populate their profile information, such as Facebook, could request our user data to streamline one of their user's experiences with their own application by integrating information that has already been inputted by said user.

3. **Data and Schema**
   - *Please see below for the full ER diagram.* The idea is that user, group, playlist, and song are all relatively standard tables in a relational database. The most useful information is stored in an Analytics relation (table), which logs information every time a song is played. Aggregation functions can be invoked to perform analytics and gain insights into streaming trends, usage, performance, etc.
     i. **User relation:** A user would require a unique identifier (UID), a real name, type (artist, listener, admin, scientist), main_genre (N/A unless artist), and age to exist. Any other fields would be optional. Users may be in groups, owners of playlists or subscribers to playlists. Users may also like songs adding them to a default playlist under their UID
     ii. **Playlist relation:** A playlist would require a UID, an owner, a name, and a subscriber count. A playlist is owned by an individual user, and zero or more [non-owner] users can select to subscribe to a playlist. All other fields would be optional. A playlist may have zero or more songs and subscribers.
     iii. **Group relation:** A group would require a UID, an admin, and a name. All other fields would be optional. A group may have zero or more members and a description
     iv. **Analytics relation:** Each analytics entry requires a UID, a song reference ID, a playlist ID, a timestamp, and a user reference ID.

All other fields would be optional. An analytics entry may make a reference to a group id. Every time a song is played a new entry is added to the analytics relation

v. **Song relation:** A song would require a UID, name, duration, artist, lyrics, and number of times liked field. All other fields would be optional.

- Song and artist data will be sourced from [here](). We will manually add mock user, group, playlist, and analytics data to supplement this dataset as required
- [Full ER diagram:]()[1]



Part 3:

Reconsider your answers to the part 3 questions in the preliminary proposal. Then elaborate in more detail how you will test the main features of your server from the external client perspective.

We will have both integration and unit testing. Unit testing would cover things like testing averageLengthSong returns the correct result. More detail is provided below.

- Unit test types:
  - **CRUD Functions**
    - Assert that user registration leads to size of user relation growing by 1 (similar for group and playlist)
    - Assert that the playSong() function leads to the size of the song relation growing by 1

[1] Client 3 will not make use of the Analytics Entity.

        - Similar to the above, run tests that functions for adding/deleting users, groups, playlists and songs lead to each of those respective relations shrinking
        - Also, want to test that update functions lead to the appropriate field changes within the appropriate relation (e.g., update user's name should only change the name field of a user)
    - **Analytics Functions**
        - Run assertions testing functions based on a sample database that the following return the proper result
            - Most popular songs
            - Average song length
            - Least listened to songs
            - Most liked songs
            - Genre with artists that are most listened to
            - Total number of songs played (can also be in terms of total time)

After the service has been fully developed, we will perform integration testing to test the service in its entirety. For example, we could do a series of route calls since we are writing a RESTful API and check that the result makes sense for that sequence of calls. More details are provided below.

- Standard workflows (i.e., series of route calls) to test
    - Authentication and authorization flows
        - test that login route correctly registers different types of users as online and then restricts their functionality accordingly
            - ./login -> leads to listener being online -> listener tries to invoke ./deleteSong/uid=1234 should lead to no changes to song relation
    - Analytics flows
        - Test that a route for returning data for something like a data scientist dashboard works as intended
            - ./login data scientist -> ./dataBoard/uid=1234 -> returns a series of JSON objects (or something like a JSON object) with appropriate information
            - ./login artist -> ./artistInfo/uid=1234 -> returns information pertaining to a specific artist like how much each of their songs has been played, how many playlists have each song etc.
    - Communications flows

        - Test that if a playlist is subscribed to and that that playlist then changes, subscribers receive the updated playlist
            - Login listener who is owner of playlist -> ./updatePlaylist/uid=1234 -> modifies playlist -> exit system -> ./login as listener subscribed to playlist -> ./getPlaylist/uid=1234 returns the correct information

Part 4:

List the libraries, frameworks, tools, etc. you think you will use. Make sure to include (at least) a build tool, a style checker, a static analysis bug finder, a test runner, a mocking framework, and a coverage tracker suitable for your language and platform. You may also choose a continuous integration tool, but github's built-in workflow actions are recommended. You will not be held to this list.

Libraries (besides others listed below): Spring Security, Connection and DriverManager for PostgresSQL, Apache HTTP Client
Framework: Spring Boot
Build Tool: Maven
Style Checker: Checkstyle
Static Analysis Bug Finder: IntelliJ
Test Runner: JUnit
Mocking Framework: Mockito
Coverage Tracker: IntelliJ