# Deep State

## Overview

Deep State is an app that broadcasts your personalized, live, always-on radio station featuring your favorite music. You can DJ your station live, or program it to play music automatically according to your tastes. You and your friends can listen to each other's stations live in realtime.

This document describes a minimal feature set and requirements to prove out the idea for the app.

## Approach

The design is premised on users having a paid third-party service for music catalog access and playback capabilities. Spotify and Apple Music, for example, offer APIs that could meet the needs of the app.

The service operates on a "tree in the forest" principle. As long as there are listeners, your station will be constantly updated. When no one is listening, we won't bother keeping a playlist going for your station. As soon as a listener shows up, the station gets going again.

The app is serverless, using a real-time shared cloud database like Firebase's Realtime DB or Cloud Firestore (or whatever) with all responsibility for modifying shared objects distributed among clients.

## Platform Requirements

The minimally viable app runs natively on iPhones running iOS 11 or newer and requires a paid Apple Music subscription (or free trial) in the US region. This lets us assume that all users have access to the same Apple Music catalog.

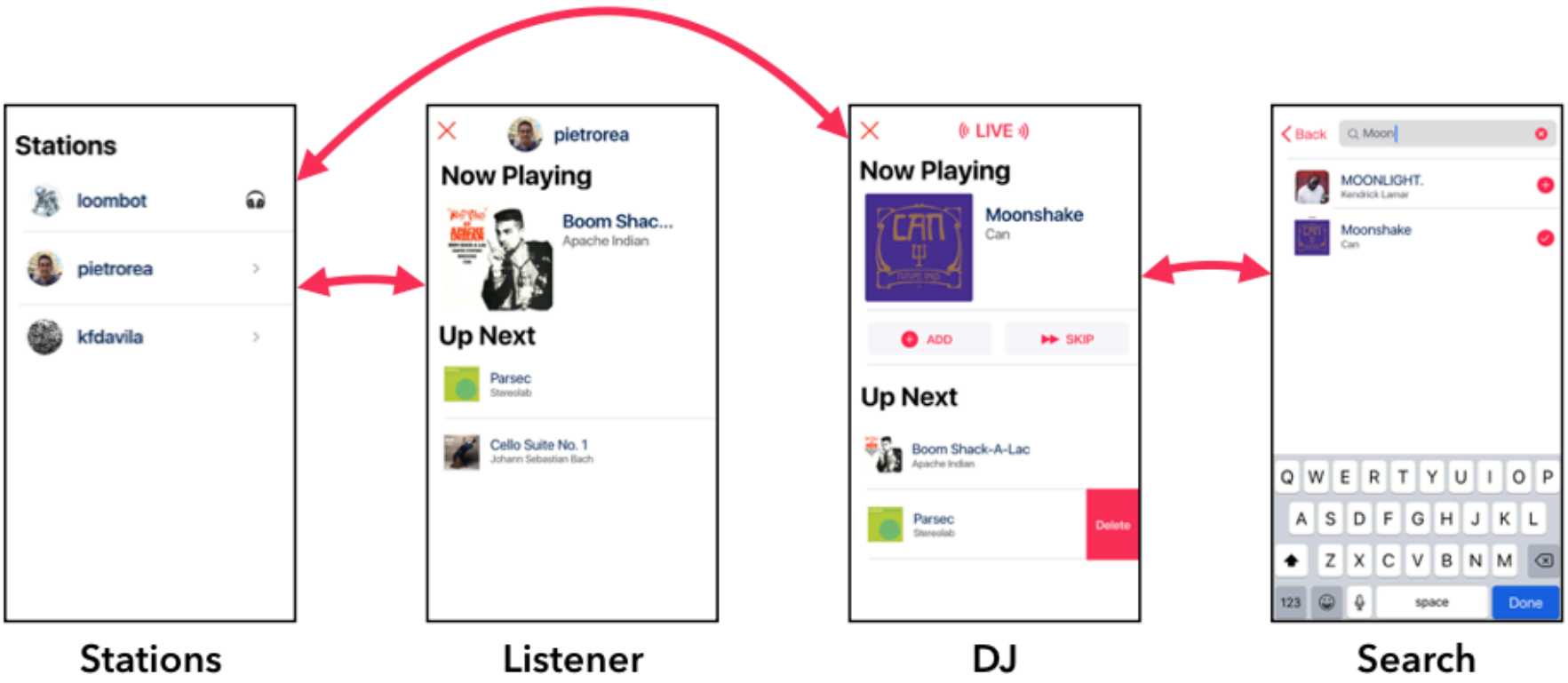
The app uses Twitter for authentication and social connections.

In theory, in future, the application could be fully cross platform:

- Spotify offers its native SDK for Android and iOS, as well as APIs for web playback
- Apple Music has an iOS SDK, and and now MusicKit JS for web playback
- Songs can be resolved between catalogs (theoretically!) via ISRC, a universal identifier for music recordings, supported by both services

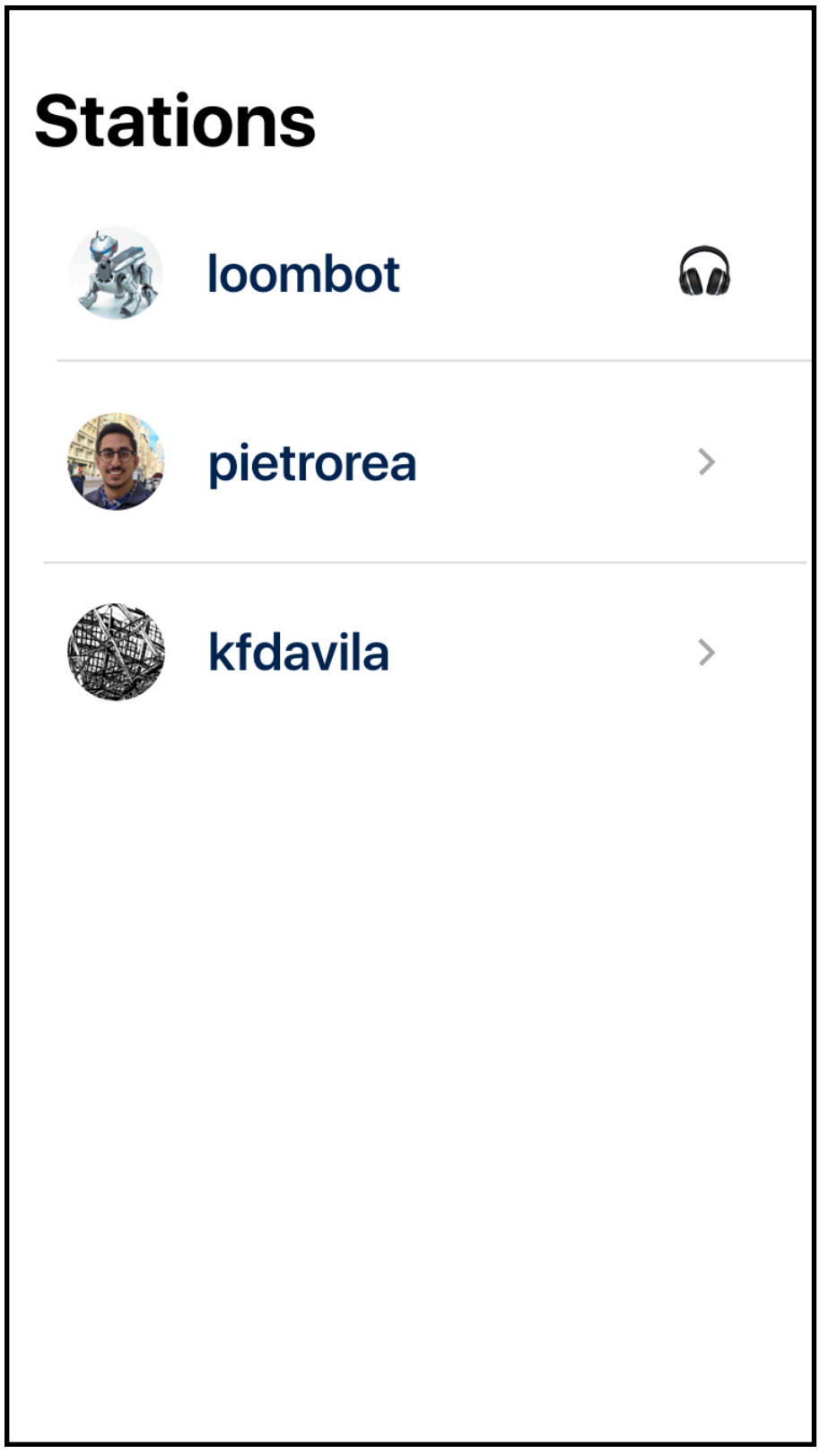
## User Experience

Not covered here: The initial authorization screens for your Apple Music account and for Twitter. Just assume we do the simplest, jankiest authorizations on startup for now.




We describe the user experience in four primary screens.

## Stations Screen




The user starts their session here. The first station on the dial is the user's own station, and tapping on it leads to the DJ screen.

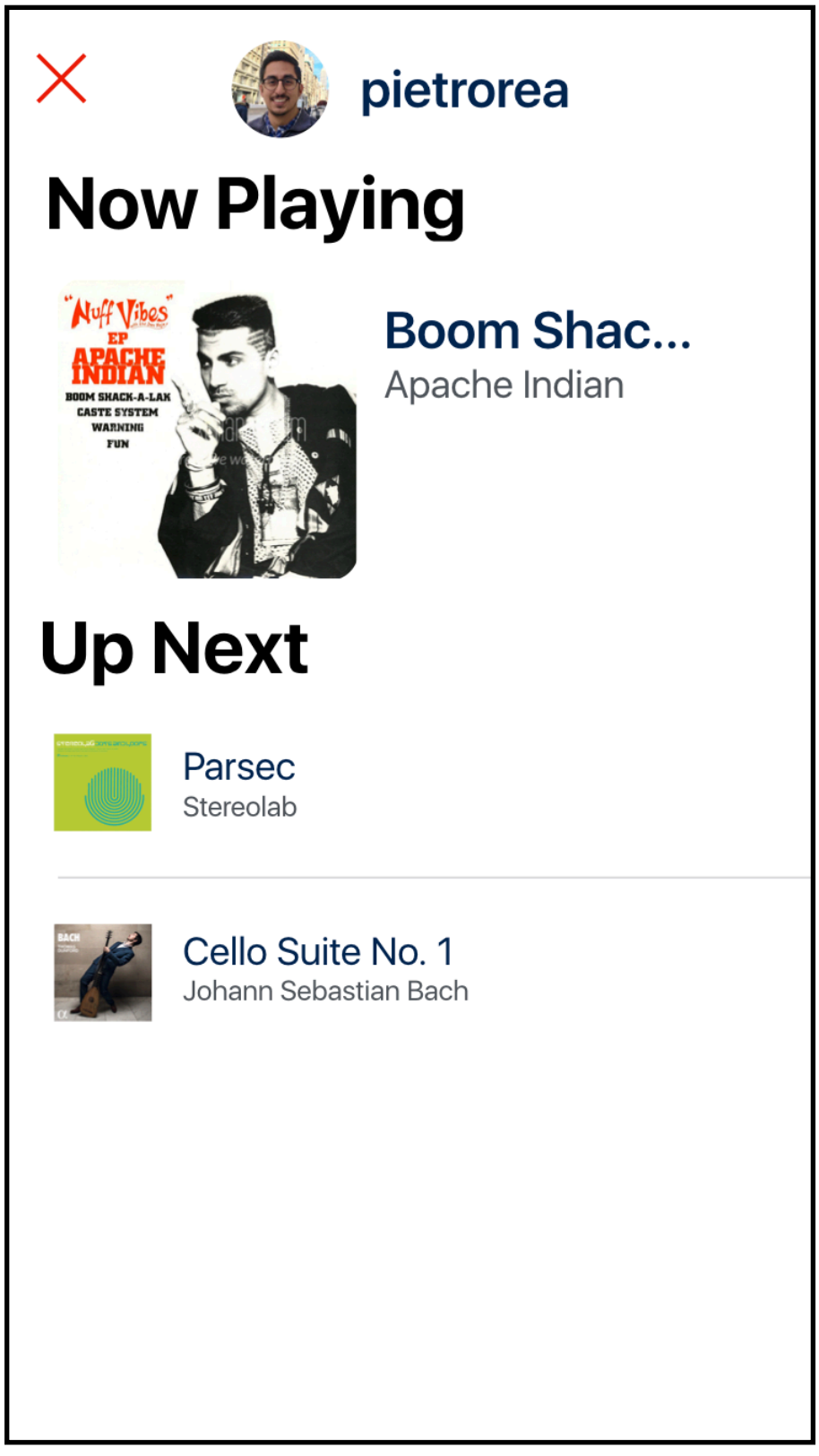
All the other rows are accounts you follow on Twitter. At the top of the list are users who have registered with Deep State. Tapping any one of them leads to listener mode.

The rest of your Twitter contacts are also shown on the list, even though they haven't registered for Deep State. We will still create stations for them, more on that later.

Specific ordering of these two sub-sections of the user list TBD.

The stations screen is managed by the user service, detailed below.

## Listener Screen

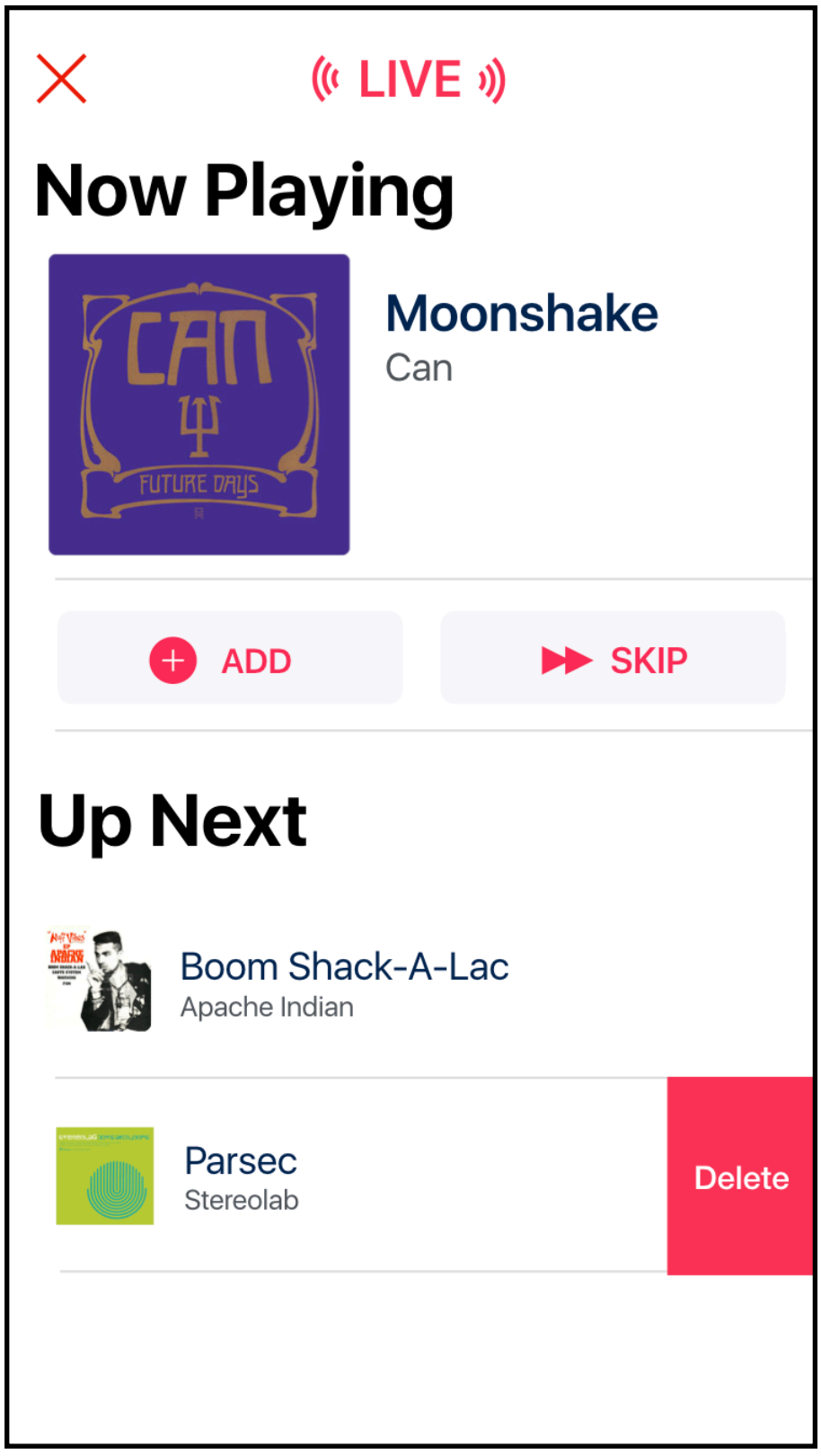



Arriving on the listener screen immediately begins playback of the station. Leaving/closing the screen stops playback.

We show the user the album art of the current track, and the songs up next. That's about it.

Actual playback duties are handled by the playback service described below.

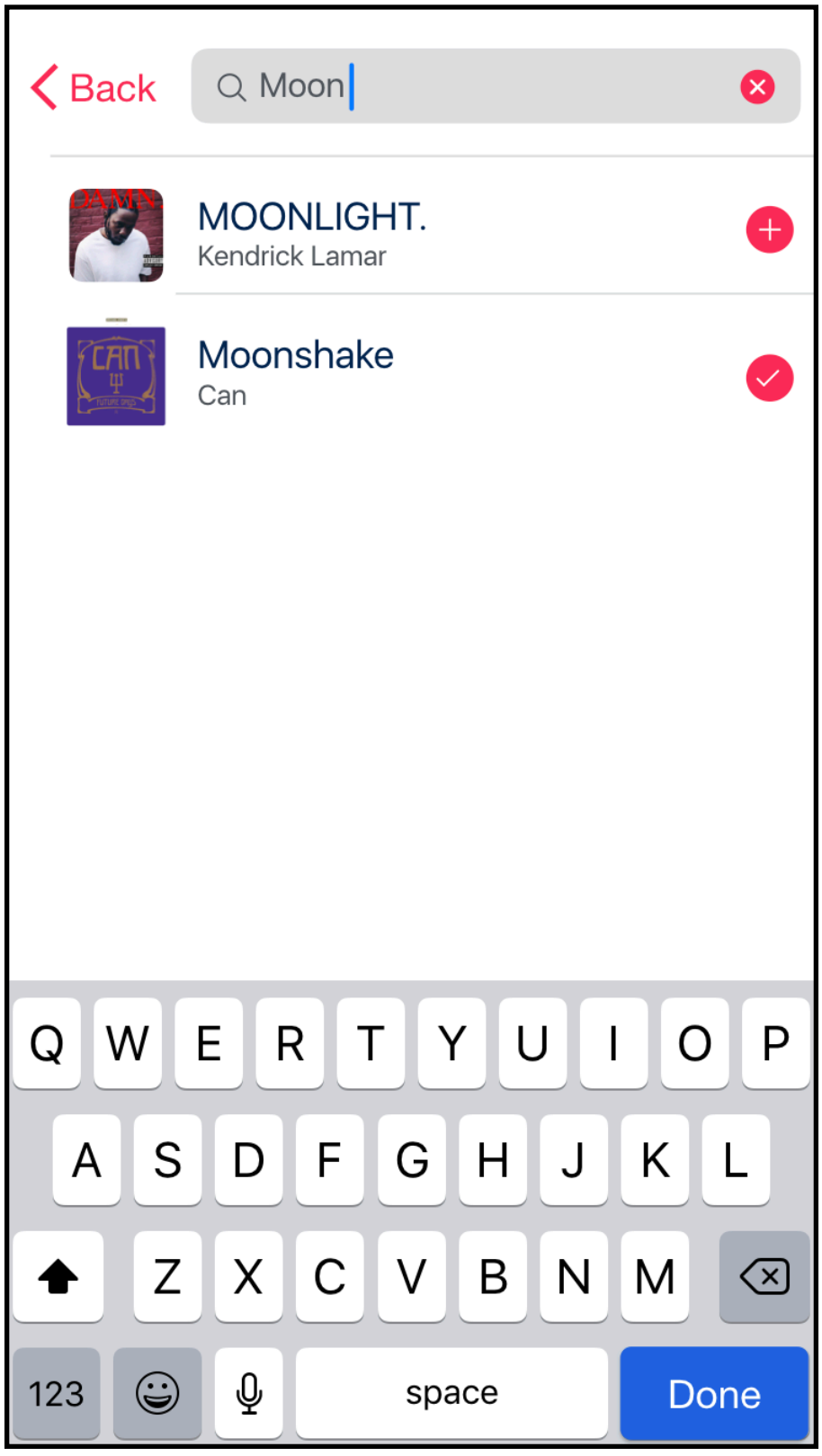
## DJ Screen




The DJ screen behaves like the listener screen in that it automatically begins playback when the user arrives there. It uses the same playback service behaviors as the listener screen.

Additionally, the DJ can perform the following tasks: Searching for and inserting tracks into the station's playlist, skipping the current track, or deleting upcoming tracks from the playlist.

## Search Screen




The search screen is a sub-screen of the DJ screen and offers a find-as-you-type interface to query the music service catalog, and to insert any track in the query results into your station's playlist in the "Up Next" position.

Tapping the "Plus" toggle next to a result adds it to the playlist. Tapping it again removes it from the playlist. Hitting "Back" returns you to the DJ screen. You can add multiple songs to the playlist without leaving this screen.

## Data Model

A proposed approach pseudo-modeled in a conceptual hierarchy. Imagine this fitting into Cloud Firestore data structures.

- Users
  - loombot
    - loombot's song pool
      - song 1
      - song 2
      - song 3
      - song 4
    - loombot's friends
      - pietrorea
      - bradegawa
      - kfdavila
    - loombot's playlist
      - 0: song 3
        - song 3's expiration time
      - 1: song 4
        - song 4's expiration time
      - 2: song 1
        - song 1's expiration time
  - pietrorea
    - pietrorea's song pool
      - song 2
      - song 3
      - song 4
    - pietrorea's friends
      - loombot
    - pietrorea's playlist
      - 0: song 3
        - song 3's expiration time
      - 1: song 2
        - song 2's expiration time

Most of these items should be self-explanatory, and the details of how they're modeled may vary depending on the data store that is chosen.

Some of the non-obvious elements are:

`user's playlist`

> A first-in first-out queue of songs. The playback service populates the queue, and the DJ can manually add to it.

`song pool`

> The pool of songs that are available to add to the station by the robot DJ service. The pool is populated by the user service.

`song's expiration time`

> When a song is added to an empty playlist, we give it an expiration time, which is the current timestamp plus the song's duration. When a song is added to the end of a playlist, its expiration time is the expiration time of the song ahead of it in the queue plus its duration. Clients check this expiration time to decide whether to play the song or remove it from the playlist. See playback service section of this document for more details.

## Service Model

Here we lay out the conceptual functions of four proposed services that support the above screens and support UI and playback behavior.

All services live and run in the client app. They run as long as the app is foregrounded, and also in the background whenever audio is playing.

Wherever we expect a service to be able to observe the state of a shared resource, like a station playlist, we are relying on the datastore's sync layer to push updates made locally and notify us of updates from remote clients.

## User Service

The first responsibility of the user service is to populate the stations list. The service should query Twitter for the user's following list at the start of a new session (app returning from background). It should group the user's stations list with users who have registered on the service at the top, and all other Twitter accounts below.

The second responsibility of the user service is populating the user's song pool. We do this at the start of a new session, up to once per day. We pull songs from the following Apple Music APIs and add them to the pool. (We could in theory add them to the pool with a TTL depending on whether the chosen data store supports that.)

- Heavy Rotation

- Recently Added
- Recommendations

Some Spotify APIs that could also be used:

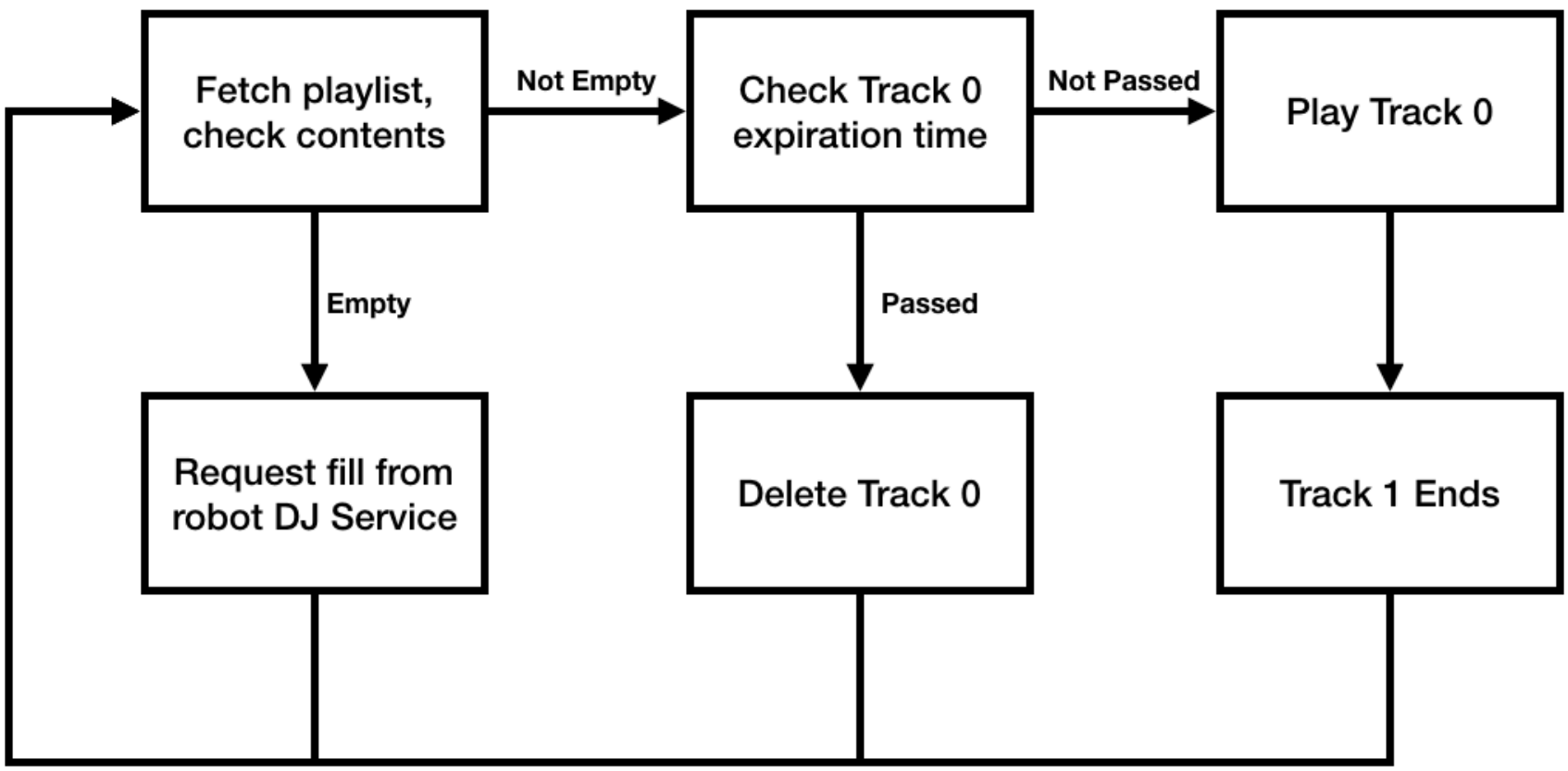
- Seeded Recommendations
- User's Top Tracks

## Playback Service

The playback service is the most important service. It fetches and monitors the current playlist, requests playlist fills, removes songs from the front of the queue, and controls local media services playback on the user's device. It runs on the listener and DJ screens.

When the user opens the DJ or listener screens, we immediately check the current playlist for the selected user's station, and subscribe to updates to the playlist as published by the datastore.

Collectively, all clients running the playback service control the state of the playback queue. The basic playback loop revolves around inspection of the first track (track 0):




In this flow, the client listens to all of track 0 once playback has started. After the track ends, the client evaluates the current "track 0" in the playlist, and either plays it or removes it

from the front of the queue. If track 0 changes *while* the client is listening, the client doesn't care. It doesn't interrupt playback.

The playback service is always subscribed to the latest state of the full playlist (so it can keep the UI of the listener screen up to date), but really only cares about its two possible states: playing the current track, or stopped and evaluating the track in the first position in the playback queue and either playing it or deleting it.

APIs to enable the playback service:

- Apple's System Music Player can play tracks from Apple Music and notify when a track has finished playing
- Spotify's player does the same for Spotify tracks
- Note that both services nominally support ISRC (a universal music identifier, see Apple, Spotify), which could in theory allow for playback across services

Note: Depending on the capability of the playback APIs, we could start playback of the first track at a time offset calculated from its expiration time. I believe Spotify supports this, I don't think Apple does, but I'm not certain.

## DJ Service

While on the DJ screen, the user is getting the same experience as listeners (i.e. playback starts automatically based on the above described logic), but has three additional functions it can perform.

First, the DJ can skip the current track. This action will remove the track in position 0, or the front of the queue, from the playlist. For the DJ only, it will also skip playback of the song and immediately move on to playing the new song in the first position, according to the regular playback service logic.

Second, the DJ can delete any track from the upcoming playlist, including tracks added by the robot DJ.

Note that the skip and delete operations require updating expiration times on subsequent items in the playlist.

Finally, the DJ can search for and add tracks to the playlist. Tracks are always inserted to the last position on the playlist. When a new track is added to the playlist queue, it is given the appropriate expiration time, taking into account the track's duration, and the expiration time of the track ahead of it in the queue.

Relevant search APIs:

- Spotify
- Apple Music

## Robot DJ Service

The robot DJ is a naive service running locally on the client, though you can imagine one day a world in which it is both more sophisticated and runs as a remote service. It can only be invoked by the listener service.

In short, it inspects the current playlist, and if the playlist is empty, it fills it with tracks chosen at random from the user's song pool, until the playlist is "full," i.e. has reach a certain total duration.

Whenever a track is added to the playlist, it follows the same rules we've specific elsewhere for its expiration time: current time plus track duration when the playlist is empty, or the expiration time of the last track on the list plus the duration of the current track.

If no songs are available in the user's pool, it picks from a special pool that will be filled manually. This is how we will spoof stations for Twitter users who haven't actually used Deep State.

## Future Considerations and Notes

A lot of the service's potential lies in the automated programming of the station. Obviously a non-naive algorithm could produce more interesting content. More feedback from both the DJ and the listeners could also result in higher quality station programming. I can even imagine an algorithm that would create thematic blocks of music as "shows." We could also insert our own content between songs. (Music service licenses would preclude paid ads but not other original content.)

Given a “Now Playing” API you could in theory sync the user’s station to music they are currently listening to, even *outside the app*. So my regular iTunes sessions could become live broadcast sessions to any other user interested in hearing what I’m hearing.

You could even let users record their own messages to play between songs.

Other ideas: Adding songs you hear to your Apple Music library, and adding songs from your Apple Music library to your station. Adding albums. More song information for listeners. Listener counts and feedback directed at the DJ.

An idealized experience would start playback in the middle of songs, and also let you quickly “flip” between stations, more closely mimicking a terrestrial radio experience.

The app should support proper integration with iOS control center to allow the user to stop the music from outside the app.

## See Also

This app is inspired by the defunct turntable.fm.

Stationhead uses a very similar approach, and even includes a collaborative voice clip feature. They’ve gotten some press.

A Spotify engineer’s hack to deliver a collaborative (voting-based) listening experience.