# iOS Music Requests


<jcapps@stanford.edu> James Capps
<debaetsa@stanford.edu> Alex De Baets
<maxrader@stanford.edu> Max Radermacher
<mvolk@stanford.edu> Matt Volk


## Project Description

We are building an iOS app that automates playing music for a group of people. The application allows the group of people to pick the music, allowing all listeners to have a say in what's playing. The application broadcasts the list of songs in a library, allowing nearby users to make requests, upvote, and downvote songs. The broadcasting device consolidates this information, using it to choose which songs to play. This product can be used in a variety of situations; we currently imagine it being used in smaller environments within a house or an apartment, but in the future, we are going to expand it to make it practical for use at organized events (like prom) or by independent radio DJs.

### Application Operation

The application has two operating modes: "host" and listener. The host operates as both the host of the party and the host in the client-server model. This device plays songs, and it broadcasts a list of songs in its library to allow the listeners ("clients") to make requests.

When operating in "host" mode, the user can configure various settings. The application contains options to control which songs are available for selection, the name of the broadcast, and whether or not the broadcast should be protected by an access code. The host can choose to limit the song selection to a particular playlist or can simply make "All Music" available. The name of the broadcast is the name that will appear in the list of "Remote Sessions" for the listeners. This list is comprised of all of the broadcasts that exist on the local network. Because we operate under the assumption that all devices are on the local network, we do not need to add any notion of an account to the application. Everything works without needing to authenticate particular users. To protect a particular broadcast, an access code can be specified; this must be entered by listeners to see the list of upcoming songs and make requests.

Listeners connect to the host device to receive the data for the application. The host sends both the list of upcoming songs and the songs available for selection within the library. Listeners can then make requests or upvote/downvote songs. These choices are sent back to the host, which consolidates the information to help sort and select songs. (In addition to the information from clients, information such as how recently a song was played will be factored in to the decision about how to sort the list of upcoming songs.) The host can let the music play in that

order or can override the queue to have a custom order. The listeners see a list of upcoming songs without the number of requests, so it does not appear strange if the host reorders the queue.

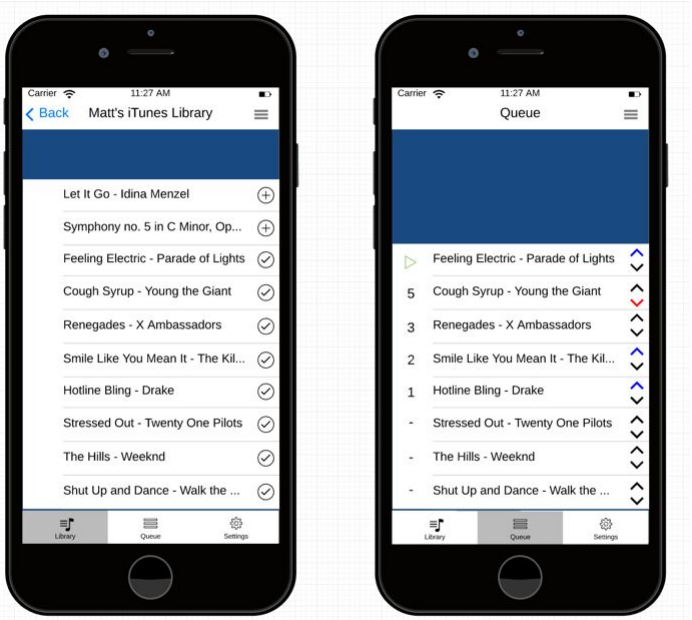


Figure 1(a): Library

Figure 1(b): Queue

**Application Pages**

The application is divided into three main pages: the queue, the settings, and the library. The queue is the default and primary view—the others serve to assist it.

- **Queue / Now Playing**

  The queue shows a list of which songs will be played next. For the host, there are additional controls to play/pause, skip songs, change the volume, reorder the queue, and view the number of requests for each song. The listeners simply see the queue without controls to modify it, nor do they see the play counts. Figure 1(b) shows a view of this page.

- **Settings / Sources**

  The settings page shows a combination of two things: the list of settings for controlling the broadcast from your own device, and a list of the available remote broadcasts in the area. It makes sense to combine these two screens since your broadcast is meant to be thought of as just another entry in the list of broadcasts. (What makes it unique is the addition of all the configuration options.)

- **Library**

  The library page shows a list of all the songs available for selection. The host has the choice to view the entire list of songs on the device or just those selected for broadcasting. The listeners will only see the songs made available by the host. The page

is divided into sections that allow selection by artists, album, song, etc. Choosing songs on this page is equivalent to upvoting songs already in the queue. (In other words, it will add them to the queue if they are not there, or it will give them an upvote if they are already in the queue.) Figure 1(a) shows a view of this page.

**Stretch Goals**

The following features are stretch goals, listed in order of implementation.

- **Bluetooth Communication**
  Communication using Bluetooth allows the listeners to connect and request songs without needing join a Wi-Fi network. This is helpful both in situation where the network is unreliable or unpredictable (as is the case with larger networks like Stanford's) or when there is no network at all (like when in most cars).

- **iPad / AirPlay View**
  We are designing the application to support all screen sizes. However, there are some iPad-specific features (such as split-screen multi-tasking, the ability to show multiple columns of content simultaneously, etc.) that we will implement if possible. In addition, we can provide extra content to display on an external display if one is attached.

- **Spotify / SoundCloud Integration**
  We are starting by using the built-in API and music library stored on the device. Apple provides an easy-to-use framework that allows us to focus on building other aspects of the application. If time permits, we are going to explore incorporating songs from other popular services.

- **Multi-Device Playback**
  For users who are not in the same physical location but would like to make a playlist together, multi-device playback will allow users to sync their audio over the Internet.

## Need for Product

Our product automates an essential task for organizing a party: handling the music. The host, once constantly interrupted with song requests, is now free to sit back and enjoy the party. The playlist is free to adapt to the mood as people's musical interests change. Moreover, no additional hardware is needed—this works with the devices people already own.

## Potential Audience

The audience for our application is iTunes users trying to create a collaborative playlist where songs are queued in real time based on listeners' preferences. This application is perfect for

hosting gatherings, from small groups to large parties. In the future, this application could also be used by independent radio DJs (such as those at their college station) to allow requests during a radio show. The host must have music stored locally on their device, but listeners are not required to have music locally. This application works even if not everybody has the application. For example, even if only several individuals in a larger group have it, they can add requests from other people in the area. Finally, no extra technical knowledge is needed—the application is designed to be easy to use for anyone who has used iTunes.

## Similar & Competing Products

Our application is roughly based on an old, discontinued iTunes feature called iTunes DJ.

- **Partify.club** is a chrome extension that controls Spotify. The extension must be run in the desktop version of Chrome, and so the host device therefore must be a traditional computer. Our application uses an iOS device as the primary server for the party, and therefore completely removes the need to have a computer involved. In addition, using Partify.club requires users to install Chrome, a Chrome extension, and Spotify, in addition to configuring it to work properly. For the average computer, this is way too complicated to be practical. In addition, some people choose to not use Spotify.
- **Track.tl** is a similar collaborative music player. However, Track.tl requires the host to sign in via a social network account. In addition, Track.tl forces the listeners to follow a unique URL to the "party" in order to join (rather than using location or a wireless network). Furthermore, Track.tl pulls its music from Deezer, YouTube, and SoundCloud.
- Both **CrowdSound** and **QCast** are very similar to our application, but both applications require authentication via a social media or email account. In addition, CrowdSound and QCast both populate their music libraries using Spotify and other web streaming services. Neither application allows the host to use his or her music library.

## Major Technologies Used

At a very high level, we are implementing this project within the realm of an iOS application (theoretically supporting iPhones, iPods, and iPads with the same binary). As such, many of the choices for technologies have been guided by the requirements imposed by Apple. When we did have a choice of a few technologies, we tended towards those recommended or endorsed by Apple since they have the best support within the platform. We tried to stay within a single platform (i.e., no web service) to reduce the complexity of the project.

As for an IDE, much of the development is taking place within Xcode because of its close integration with the platform. In addition, the supplemental tools provided as part of the package (Interface Builder, iOS Simulator, Instruments, etc.) are being used to aid in the development.

**Swift**

Apple officially supports two languages when developing a native application for iOS devices: Swift and Objective-C. While the application can be written in another tool and converted to native code later, Apple does not recommend this approach. Because of the generally lower quality that these tools produce, and because of the fact that members of the team wish to learn iOS programming, we are using Swift, one of the official languages.

While no members of the team have much experience with Swift, it is the only language needed by the project. (We don't have a web component, so we don't need to worry about server-side languages.) Many free resources exist online to aid in the learning process for Swift. (Plus, the language is syntactically similar to current popular languages, so it won't look nearly as foreign as something like Objective-C.)

**Music API**

A major reason for choosing to build an iOS app is the predictable, closed nature of the platform. A closed, controlled environment makes it reasonable to develop an application in the allotted time. Along with this, because the devices are predictable and consistent, Apple is able to provide easy-to-use APIs that are "guaranteed" to work. While they may not always behave as expected, they will always work without the user needing to configure anything.

One of these APIs is the iPod Music Library API, which contains two components that we are utilizing heavily in our application. The first is the ability to read the metadata for the songs contained in the library. We use this information to populate our own user interface. Apple provides everything from the name of a song to the album artwork. The second component is the ability to pass one of these song objects back to the framework to play it. The focus of our app falls between these two steps: how do we pick the next song to play?

**Bonjour / CFSocket / NSStream**

A critical component of what we are building involves communicating among multiple devices. We need to be able to send the list of songs to client devices, and we need those devices to be able to send song requests back to the main device. We'll do this using a typical reliable communications networking stack:sockets, streams, TCP, etc.

For iOS development, we are using Bonjour for the discovery of devices on the local network, the CoreFoundation layer to create the socket, and the NSStream subclasses for the actual reading and writing of data. These technologies make it easy to discover devices, but they still give us the ability to use a higher-level class for the actual communication.

**Core Data**

For the internal data model representation within the app, we are using Core Data. While this framework is generally used for persistent storage of data, some of its tools will be helpful in developing and maintaining are in-memory-only data model. Plus, it provides the team with more experience in another important iOS API.

Core Data provides a visual modeling tool to establish the relationships between various objects. Xcode then automatically generates and implements the corresponding class files. Not only does this approach avoid mistakes made when writing source code by hand, but it keeps the implementations consistent with best practices.

**Git & GitHub**

We are using Git for source control. Everyone is familiar with Git, and so we are able to use it without too much overhead. We have already discussed some of the common usage patterns for Git, and we have made decisions about how to make the best use of this tool when developing the project. We also have a public GitHub repository for the project.

## Resource Requirements

Our project requires very few resources. Because Xcode will only run on a Mac, our entire team needs access to a computer running OS X. All team members either own a Mac or have access to a cluster computer running OS X. In addition, we need to be able to test our design and layout on all of the various iPhone screen sizes. Members of our team own most of the screen sizes, and we can use the simulator for the missing screen sizes.

## Potential Approaches

We discussed a few different ways to implement our idea of "let people request songs," and we settled on our current approach after considering some of the weaknesses or potential roadblocks with the others. The main discussion points involved the source of the music and the destination(s) of the music.

- **Source**

  With respect to the source of the music, we were divided about whether or not we should source it from the local music on the device or an app like Spotify. The team was evenly divided on this issue, but we chose to use the local music given that (a) Apple provides an easy way to do so and (b) we could not find any officially supported way to accomplish something similar with Spotify. To complete the project using another service would have required more "hackish" workarounds—something we did not want to pursue.

- **Authentication**
  In testing some of the competing applications, the need to log in to an online service before playing songs was a rather intimidating barrier. By using iPod Music Library API, there is no need to log in to any account, and there is no need to register. That will make it very easy for people to quickly request songs.

- **Destination**
  For the destination of the audio, we discussed whether or not it should play only on the host device, or whether it should be possible to simultaneously play the same audio from multiple devices. The use case in the latter would be people who are physically distant (and perhaps communicating via some video chatting service) who all want to listen to the same songs. Once again, we decided to not pursue this for complexity reasons (it's more difficult when working with a local library) and copyright reasons (it's probably illegal to have our users "share" songs in this way, even if they aren't stored on other devices). Therefore, we will reconsider the topic of multi-device playback, time permitting.

- **Primary Device**
  Another design consideration is where the primary application should run: a computer vs. a phone. Running it on a computer would solve some of the roadblocks caused by the closed nature of the iOS platform (this is how Partify is able to operate) In addition, the original inspiration for this project—iTunes DJ—was a part of the Mac incarnation of iTunes. We chose to design this application as a mobile application rather than a desktop application for two main reasons. First, we are in the "post-PC" era, meaning that more and more usage is moving from traditional desktops to mobile devices. More and more often, someone will plug a phone, not a computer, into speakers to play music. Second, our team is more interested in learning iOS development than Mac development.

**Assessment of Risks**

While building the application, we need to be aware of risks and possible obstacles, several of which are listed below.

- **Wi-Fi**
  As designed, the application requires that all participants are on the same Wi-Fi network. This puts a significant restriction on how the host and listeners use the app.

- **Compatibility**
  We want our application to be compatible with all modern iOS devices (iPhone, iPod touch, iPad, etc.). However, handling these various screen sizes (and properly testing

them) will require a decent amount of time and the use of hardware that we do not currently own.

- **Design**
  Another foreseeable risk lies in the design of the application. To achieve a pleasant experience for the user, the UI needs to be simple enough for easy and intuitive use, yet it simultaneously needs to be unique and non-generic to be appealing. In designing the app we will have to balance these two requirements.

- **Trend**
  In building this application, we recognize that one of the greatest risks is the decreasing population of users in the iTunes ecosystem. As technologies and forms of entertainment shift and change, the application may because incompatible and lose popularity. One example is the fact that users have started moving to other services (such as Spotify).

## Next Steps

In the coming weeks, we will continue to work diligently on our User Interface. Once we have agreed on a basic user interface, we will work together to lay the groundwork for the basic backend design and implementation of the application. Once this basic framework is completed, each team member will work on separate individual tasks within the branches. There are four main branches of this model: User Interface (including layout and graphic design of the app), Networking (handling Wi-Fi requests and managing the queue of songs), Importing Data (retrieving song data from iTunes), and Storing Data (implementing how the retrieved data is represented). Each team member will begin working on part or multiple parts of this model.