Pick Play Music Readme

Adam Raudonis, Evan Plotkin, Kyle Dumovic, and Ranajay Sen

Native iOS App Read Me

Installation Instructions
We are distributing the native app over testflight. We can invite anyone to download the app if they email us their UDID.

Frameworks

The Cocoa Lib Spotify iOS
Spotify's framework is used for getting track metadata, album artwork, real-time searching, and most crucially playing songs. It has some special requirements however, which is it would not work with the default arm64 Build Architecture, so I needed to change the Valid Architectures to be armv7 and armv7s.

QREncoder
This is used when the device is in lock mode to display a QR Code as the Album artwork that links to web app. However, this naive implementation of just setting the UIImage returned in the QREncoder Framework as the MPMediaPlayer album art did not work. This is because QR Codes require a white border around them. I then used a third party UIImage+Resize category to resize the QR-Code to almost 1000x1000 pixels so it would look crisp on retina displays. I then created a new blank UIGraphicsImageContext that was slightly larger. By using the CGGraphic's drawInRect method I then put the QR-Code centered on top of the white canvas creating a border. Finally through a call to the system method UIGraphicsGetImageFromCurrentImageContext I had a usable QR-Code.

AFNetworking
An awesome, easy to use, fast wrapper for network requests.

MPProgressHUD
Used for locking the UI when important network events are occurring so the user

cannot accidentally click for example “Create”. It then displays messages such as loading playlists.

Code

My code is broken up in three main parts View Controllers, Global Code and Models.

GroupPlayViewController

This is the “homepage” of the app. It includes a UITableView that lists the user’s current rooms they are a part of. This data is generated through a call to searchRooms passing in the parameter of the current user id which is stored in NSUserDefaults.

CreateRoomViewController

The user is prompted to log into spotify with a default spotify login view upon clicking create room in the groupPlayViewController. If the user has a spotify premium account then they are allow to proceed to the createRoomViewController where can set the name and optional password of the room with UITextFields and whether everyone can edit and the mode with UISwitches. The VC also determines the user’s current location. It continually checks for the location while on the page to increase accuracy. It stops updating when the user clicks next. The room is not actually created at this point.

SeedPlaylistViewController

This VC enumerates all the user’s spotify profiles. Users can check multiple uitableviewcells and that will do a union of all the SPPlaylistItems (essentially SPTracks) and pass those on as initial songs for the queue. My group wanted me to impose the arbitrary limit of 50 songs, so I enforced that constraint. Upon clicking next, the server then create the room and get the playlist (so

QueueViewController

This is the main VC of the app. It contains a UITableView that displays the song queue. It also contains in app pause, play, skip buttons and logic for setting the MPMediaPlayer song info so that even outside the app you can still have all of that functionality.

It also contains the logic for taking a dictionary of track meta data and spotify url and builds SPTrack objects out of them using the Spotify Library. These are then able to be played by the SPPlaybackManager. While the song is playing I use key value observers to update the track current position and listen for the end of the song. Every couple of seconds I poll our own server and get the play queue. That way if someone using the webapp reorders a song, it will be reflected in close to real time. Once the song is finished, I then update the song queue on the server all the web apps know the current song playing.

Initially we planned on having a Settings UIBarButtonItem in upper right of the

UINavigationBar. This meant that there was no place to put an edit button. I thought the next best alternative would be to have swipe to delete (without an icon) and drag to reorder also without an icon. While frameworks existed to address each problem individually, no one had combined the two publicly. I heavily modified the reordering framework https://github.com/FlorianMielke/FMMoveTableView along with the swipe to delete framework https://github.com/TeehanLax/UITableViewCell-Swipe-for-Options to create my combined solution: https://github.com/adamraudonis/UITableViewCell-Swipe-for-Options Sadly this component was not used in the end product.

Finally, the QueueViewController is also responsible for implementing search. There are two different ways that search works. The first is through the Spotify framework's native method. The second is using the spotify metadata api. This is because I made it so the iOS app is still usable to join rooms even if you don't have a spotify premium account.

JoinRoomViewController
In this VC, on viewDidLoad I create a CLLocationManager instance and start updating the current location of the user. This gets returned to me through the CLLocationManagerDelegate delegate methods. I will let this method be called up to 3 times before stopping updating. The reason for this is that the returned location may be inaccurate in the first call. Subsequent calls have higher and higher degrees of accuracy. I found there were diminishing returns after 3 calls. After each location call I will poll the server using the searchRooms call to list the closeby rooms in a UITableView. If you tap on a tableViewCell you will either perform a segue to the QueueViewController or the PasswordViewController depending on if the room has a password.

PasswordViewController
This is a simple VC with one UITextField. If the user types in the correct password and then hits enter a call is made to join rooms. If join rooms is successful then a segue is performed to the queue. If an error status corresponding to an incorrect password occurs then that is displayed to the user and segue not performed.

User
This class abstracts a user object. It provides methods for easily creating it from a NSDictionary

Room
This class abstracts a room object. It also has init methods that are custom for creation.

Utils
Contains various helper methods that are commented below.

```
// Returns the user stored in NSUserDefaults
+ (User *)storedUser;
// Returns the base url for the server
+ (NSString *)endpoint:(NSString *)endpoint;
// Returns an image with a color border
+ (UIImage *)imageWithColor:(UIColor *)color image:(UIImage*)image;
// Returns an image of a webapp url qrcode to a room
+ (UIImage *)qrcodeForRoom:(Room *)room;
// Shows an alert view for a server error
+ (void)alertForServerFailure:(id)responseObject;
// Converts an NSObject into a JSON string
+ (NSString *)jsonDumps:(id)object;
// Shows a UIAlertView
+ (void)errorAlertWithMessage:(NSString *)message;
// Stores a password
+ (void)storePassword:(NSString *)password id:(NSString *)room_id;
// Retrieves a stored password
+ (NSString *)storedPasswordForRoom:(Room *)room;
// Returns the name of a random fruit.
+ (NSString *)randomFruitName;
// Replace with _ all non-alphanum's & '-'
+ (NSString *)sanitize:(NSString *)str;
```

App Delegate
On application launch this is used to init the spotify library. This happens ever before the user has logged into Spotify. Also some app-wide UI is set here such as the color of the status bar.

Storyboard
I also use XCode's Storyboard to layout all the View Controllers. I then have a custom view controller class associated with each view controller.

Testing
Test Flight is used to ad hoc distribute the application based on mobile provisioning profiles with a specific set of users.

Backend Readme

Installation Instructions
It is possible to download the repo and deploy it as a Google App Engine app using

their launcher, but it is easier to use our version of the app already hosted online at the API listed below.

Google App Engine
We chose to use Google App Engine with Python as our framework because we were very familiar with Python, it was easy to deploy, and didn't require the additional system and server configuration that other frameworks often run into. GAE's ability to add servers as needed if we ever take on more users was also important.

webapp2
Webapp2 is the extremely simple WSGI application that routes url's to their respective Python applications. Since our backend was just a series of simple JSON endpoints, we decided to use this over more complex frameworks like Django, Flask, and Bottle.

ndb
The schemaless object datastore in GAE was suitable for our needs as we didn't need to do any complex folding or data accumulation queries. It also has automatic caching and atomic transactions, which is necessary for a multi-user application like ours is. One of our major challenges was shifting from our experience with schema databases like SQL to this datastore type with entities and models. The first part of this task involved creating data models that were suited to this sort of structure, and nesting them in the graph structure in a connected fashion that made sense (more on this later in models). The second part of this task was learning how to fashion queries and keep references to other items, and we chose to keep track of the integer version of the ids to do so.

Models
We were able to boil our entire data architecture into four main models: rooms, guests, users, and songs.

Rooms kept track of a creator, a name, a mode, a pair of coordinates, a queue of songs to be played, a history of songs already played, a boolean status whether the room is playing or not, and a password to secure the room.

Guests of a room simply kept track of the user's universal user id and a boolean depending on if they had admin status or not.

Users simply tied a unique id to a username.

Songs consisted of a Spotify url, a track name, an artist name, an album name, an image url, a boolean tracking whether the song is in history, the guest who submitted the song, and the time it was submitted.

Endpoints

We had a number of endpoints that function as per the following API guide.

Base URL

http://musicviz194.appspot.com/

Results Format

All requests should return the following json format:

```
{
        status:”OK” /* Anything other than OK will be treated as error */
        data: {...} /* or [...] */
        message: ”Error description goes here” /* Only show if error */
}
```

Check Play Status

url:/check_play_status
function: Returns whether room is playing or paused.
type: POST
params:
- -room_id

returns:
- "true" if playing, "false" if paused

Pause Song

url:/pause_song
function: Pauses or unpauses the song in a room.
type: POST
params:
- -room_id
- -action(can be either "pause" or "play")

Archive Song

url:/archive_song
function: Moves a song from queue to history.
type: POST
params:
- -room_id
- -song_id
- -user_id(of the person reordering ... must be creator)

Delete Song

url: /delete_song
function: Deletes a song from a room's queue.
type: POST
params:
- -room_id

-url : the spotify url
-position : included to prevent duplicates and check consistency
-user_id (of the person reordering … must be admin)

Delete Room
url: /delete_room
function: Delete a room.
type: POST
params:
-room_id

Delete User
url: /delete_user
function: Delete a user.
type: POST
params:
-user_id

Update Room
url: /update_room
function: Update the coordinates for a room. Use case: moving car.
type: POST
params:
-room_id
-coordinates

Reorder Song
url: /reorder_song
function: Reposition a song in a queue.
type: POST
params:
-room_id
-song_id : the unique_id
-new_pos (0 indexed)
-user_id (of the person reordering … must be admin)

Submit Song
url: /submit_song
function: Submit a song to a room's queue.
type: POST
params:
-room_id: ID that corresponds with a created room
-user_id: ID that corresponds with a guest of the room
-url: spotify url
-track : the name of the track
-artist : the name of the artist(s)
-album : the name of the albulm

Create Room
url: /create_room
function: Creates a new room.
type: POST
params:
- mode : Either test, test1, test2, or test3
- creator (id) (must be an already registered user)
- room_name
- coordinates (optional): the string lat, lng
- initial_playlist (optional) : Json array of {url,track,artist,album}
-Would it be better to do this by simply calling “submit song” upon
NOTE: Will automatically join the room for the creator
room creation?
- password (optional) : An optional password for the room
returns:
- room_id : the id of the newly created room

Join Room
url: /join_room
function: Allows a user to join a room.
type: POST
params:
- room_id
- user_id (must be an already registered user)
- password (optional)

Register User
url: /register_user
function: Registers a username with the backend.
type: POST
params:
- username
returns:
- user_id : the id of the newly created user
(Note: multiple users with the same username is ok)

Search Rooms
url: /search_room
function: Returns a list of rooms filtered with the included parameters.
type: POST
params:
- coordinates (optional) : string of lat,lng
- radius (optional) : the radius in meters for max return distance
- room_name (optional) : the name of a room
- user_id (optional) : the rooms that a user is a part of or owns
returns:
A json array of dictionaries with the following keys:
- lon

- lat
-password (boolean true/false)
- name
- creator_name
- mode

Get Song Queue
url: /get_song_queue
function: Returns the song queue for a certain room.
params:
- room_id
- num_songs (optional ... default=1)
- user_id? (only get song queue if you belong to the room?)
-type (default = queue, 'history' = history, 'both' = concatenation)
returns:
A json array of dictionaries with the following keys
-unique_id
- url
- track
- artist
- album
-song_pos (for non-history songs)

Get User From ID
url: /get_user_from_id
function: Given a user_id, returns user info.
type: POST
params:
- user_id (must be an already registered user)
returns:
- user info in json blob

Get Room From ID
url: /get_room_from_id
function: Given a room_id, returns room info.
type: POST
params:
- room_id (must be an already created room)
returns:
- room info

Modes

We decided on two major modes for the initial release:
1. First come, first serve (default) - songs are played in the order they are added.
2. Fairness - songs are played such that in each round, every guest gets to play a song of their submission before others can play another song.

User Status

Users can be one of two statuses within a certain room:
1. General - can submit songs and see which songs are playing.
2. Admin - has same abilities as guest along with the ability to delete and reorder songs.

Utility Functions
We have a number of utility functions that serve the following functions, among others:
1. A transactional put function that makes sure datastore puts are atomic.
2. Functions that find guests and users by id.
3. A function that checks whether a user is an admin of a room.
4. A custom subclassed JSON encoder that adds and censors appropriate information.
5. A distance function that using the haversine formula.
6. A function that returns a bounding box given a latitude, longitude and distance away.
7. A function that checks password hashes using SHA-512.

Security Scheme
We maintain the security in two main ways:
1. We allow room creators to secure rooms with a password. We store these passwords as secure SHA-512 hashes and check against this hash with every call to password-protected rooms so users without the password cannot see the song queue and modify anything in the room. We chose not to salt our hashes to remove overhead and because we did not think the risk of people using rainbow tables to break into temporary rooms was particularly high.
2. We allow user uniqueness and prevent "identity theft" by hiding all user_id's but your own from other users, showing them only the username (which isn't necessarily unique). Actions that affect a user like changing the username and deleting the user is only available by passing the user_id which only the user himself received when registering the username.

Interesting/Notable Work
-Deferred loading: When we create a room, we load an “initial playlist” of songs. We found that this was very slow, because for each song we would load the album art from the spotify library. So, we made use of Google App Engine’s “deferred” capabilities that would allow us to create the room and return to the user, while loading album art asynchronously.

-Fairness Reordering: Fairness mode requires that the songs in a queue are ordered in a very specific manner (round-robin by the song-submitter). To maintain this order, we must make adjustments whenever a song is inserted or deleted. The SubmitSong endpoint uses the “fairness_insert” method to find the correct position to insert a song in. This method was fairly challenging algorithmically.

Upon deleting a song, we also must reorder the song queue. When a user's song is deleted, all his songs essentially shift up one "round." The method "fairness_adjustment" in the DeleteSong endpoint ensures that proper functionality is maintained. This method was fairly challenging algorithmically.

Web Frontend Readme

Installation Instructions
Same as backend.

Chocolate Chip UI
We decided upon using ChocolateChip-UI, which I like because it is quicker than jQuery Mobile and has stylesheets that support iOS, Android and Windows Phone 8 and matches the look and feel of iOS 7. Additionally, there are great looking demos available for making lists of music and navigating between them.

Check out our app on Android, it looks perhaps even better than on iOS!

JQuery UI Autocomplete
We use the autocomplete plugin to search for songs in the Spotify library to quickly add into our queues. We use the onSelect function to provide a quick response and addition of songs.

HTML Structure
The front-end is structured into two html files--home.html and queue.html.

Home.html allows users to view nearby rooms (grabbing user location information via javascript geolocation services). They can then *join* these nearby rooms, adding them to a list of their joined rooms. From here they can *view* the respective queues of these rooms they are members of.

Queue takes the parameter of a room_id and displays the corresponding music playlist. We created a unique html file with this functionality because the iOS app has a QR code with a url that we redirect to this page.

Style Sheets
We used **SASS (Syntactically Awesome Style Sheets)** for our CSS. SASS adds a bunch of nice features to vanilla css; namely, variables (great for saving unique color combinations), mix-ins for predefined grouped CSS declarations, and nested selectors. The SASS takes a pre-processed *.scss* file and outputs it as a normal *.css* that I include in the HTML header.

Javascript
Our web app runs out of *app.js*, which is broken down into two *main()* functions, *homeReady()* and *queueReady()* which are each called on their pages' pageload.

User Registration
Both pages need to deal with new user registration that we wanted to be as painless as possible. This meant no passwords associated with user accounts, so we needed to make AJAX calls to register users with randomly selected usernames from the following array,

var username_arr = ['Banana', 'Apple', 'Peach', 'Mango', 'Cherry', 'Grape', 'Pear', 'Plum', 'Pineapple', 'Lychee', 'Kiwi'];

The unique user_ids the json returned we saved in a session cookie. Javascript makes it easy to create and save cookies with key-value pairs, but we utilized **jquery-cookie**, a simple jQuery plugin for reading, writing and deleting cookies that made it even easier. Cookies we set to expire after 2 hours, at which point visitors will be assigned new, random usernames upon arrival. The backend scrapes these now inaccessible users periodically.

We also gave users the functionality to change their randomly assigned usernames with ease. Aside from finding solace in the fact that you are no longer named after a fruit, each song in a playlist viewed in the web-app lists an "Added by [username]" characteristic that will be updated with the new preferred username.

Because rooms can be protected by passwords and users can be members of multiple password-protected rooms, we use cookies to associate hashed passwords with current user_ids.

Users who visit a password-protected room via a QR-code link will be prompted for the room password on pageload, and if the provided password is incorrect, they will not be able to view or change the room playlist.

AJAX Requests
We have the web-app send AJAX requests to the following endpoints--

Home.html
/register_user
/change_username
/search_room
/join_room

Queue.html
/get_song_queue
/submit_song
/delete_song
/reorder_song

AJAX requests have to be properly handled with callbacks and timeouts, as the HTTP requests can take up to half a second before they respond, during which time the javascript will proceed as usual, which is not ideal. This made structuring the ready functions more difficult.

Drag to Reorder Functionality

We used the **jQuery UI Sortable** feature to make our list of songs in the queue clickable and draggable for users with admin privileges (check to see if the Edit button exists at the top right of the web-app). But because jQuery only supports click devices and not touch, we used the **jQuery UI Touch Punch** plugin to add touch support for jQuery UI. The icon at the right that becomes visible upon hitting the Edit button is a handle for users to click and drag, thus maintaining the ability to scroll on iOS devices through longer lists of songs.

Notifications

We built basic notification functionality via a div at the top of the page that reveals itself and transitions (via CSS). See the *showNotification(msg)* function. We currently employ this for user notification after successfully adding and deleting songs from the queue.

Auto-reload
We make periodic AJAX calls to refresh parts of page content (rooms on home.html, songs on queue.html). We use javascript's *setInterval()* for this functionality.