Qspot

Brian W. Ferch
Prof. Louis D. Nel
Carleton University
COMP4905 - Honours Project
December 2018
Abstract

What if the host of a party wanted to allow their guests to control the music at the party in an organized way? I set out to build a web application that solves this problem by creating a “virtual jukebox” that gives party guests a way to search for songs they want to hear and add them to a shared queue that is played on the party host’s speakers. To make the system fair, songs needed to be intelligently ordered such that everyone gets their chance to have their song play. The resulting application is called Qspot. It uses Spotify under the hood to enable music playback and a large library of songs. To order songs fairly, it sorts them based on how many songs each user has already submitted. The application successfully delivers the core functionality of a “virtual jukebox” experience in an albeit unattractive interface.
Acknowledgements

I’d like to acknowledge the authors and companies whose technologies enabled Qspot to be the modern web application it is: Spotify’s API’s and SDK’s. Google’s Firebase and their suite of technologies: Firestore, Cloud Functions, Authentication. Google’s Polymer Project for their lit-element library used to construct the interface, and for their various tools and application starter kits. Rich Tibbett’s NoSleep.js library. Dan Abramov and Andrew Clark’s Redux library. I’d also like to thank my honours project supervisor Professor Louis D. Nel.
Table of Contents

1.0 Introduction 6
2.0 Methodology 8
2.1 Spotify Authentication 8
2.2 Room Creation 10
2.3 Searching for Songs 12
2.4 Music Playback 13
2.5 Queue Ordering 14
3.0 Results 16
4.0 Conclusion 20
List of Figures

2. Qspot Main View 16
3. Qspot Search View 17
4. Qspot Join View 17
5. Qspot Login View 18
1.0 Introduction

Qspot is a real-time web application that allows a group of people to listen to a queue of songs that are submitted by the users. The songs are played through one user’s (the host’s) device. Users can join a party, search for songs they want to listen to, and submit them to the queue where they are given an ordering based on the number of songs an individual has added as well as the time they were added.

The rationale behind Qspot’s creation is a party setting where the host of the party wants to give control of what to listen to to the party-goers in a fair and controlled way, and the songs are still played through the host’s sound system. Often music at house parties is a battle of who can keep jurisdiction over the AUX cable the longest. Music at clubs is subject to the whim of the DJ. Many people would prefer to share their own tastes, and explore the tastes of other people at the party, enriching social interaction.

The jukebox is an early example of letting people control the music they listen to in a social setting. Modern digital jukeboxes in bars and clubs have even been incorporating similar ideas of letting users choose their songs with an app in exchange for a fee [TouchTunes authors, 2018]. However, both of these solutions are not designed for personal use. They require heavy upfront costs for the device itself, and in the case of digital jukeboxes, subscription fees to pay for the service and licencing necessary to play songs in public spaces. Room in this sector exists to create a “personal jukebox” that anyone can easily set up and use.

The objective of Qspot is to create an application that delivers a number of key components and features that enable a “personal jukebox” experience with minimal friction, using devices and services that many people already have access to. These key components are:
A way to identify users and organize them into two distinct classes: party-goers, and party-hosts.

Party-hosts need to create a “room” in the application which organizes and groups all the songs and party-goers that belong to that party. Party-hosts need a way to give access to party-goers of their choosing to the “room.”

Once inside of a party, party-goers need to able to search for songs by entering the name/artist/album. Once presented with list of possibilities, they should be able to select a song to add it to the queue.

The party-host should be able to manually “start the party” once at least one song is in the queue. Once started, the party-host’s device should play the song at the top of the queue, and after it ends, play the next one, and so on and so forth.

All users need to able to see the queue and see it update in real-time with what’s currently playing and what songs are up next. The order of the queue should be organized by two factors: how many songs the party-goer who submitted the song had previously submitted, and when they submitted the song.
2.0 Methodology

In describing the methodology, this report seeks to treat the objectives outlined for this project as smaller sub-projects. That is to say, each of these objectives is an important and somewhat complex component of the whole application, so each will be described with its own relevant ideas, sources of information, and comparisons of approaches.

2.1 Spotify Authentication

For any of the other pieces of the application to function, some way to identify users and associate data to them was needed to keep track of songs submitted and prevent cheating. Normally this is done with an account associated to a username and password. This was one possible approach to authenticate party-goers. The other option was a social sign-in, where a social media account is used to authenticate a user. Ultimately, since part of the objective was to get party-goers using the app as quickly as possible, a social sign-in was used to avoid the potentially lengthy hassle of setting up a username and password.

Firebase provides an array of social sign-ins that can be added to an app with a simple API [Firebase authors, 2018]. Spotify, unfortunately, is not included in this array. Party-hosts by necessity needed to be authenticated with their Spotify account to be able to play music on their device. A username and password was never an option for them.

Implementing a custom authentication flow start to finish became a necessary part of the project. Authentication flows are always somewhat specific to the provider, and Spotify was no different. Spotify’s authorization guide was an extremely important document to reference for this task [Spotify authors, 2018]. It outlined two possible flows: the Authorization Code Flow, and the Implicit Grant Flow. Both flows return access tokens
which are used to grant access to privileged Spotify APIs such as the ones that control music playback for premium users. The Authorization flow is depicted visually in Fig 1.

The key thing to understand about it is that it requires step 2 in Fig. 1 to be performed from a server under the client application’s control. So a server under control of the Qspot application needs to make the requests to Spotify’s systems, instead of the request being made from the web client which runs in the user’s browser. In choosing this flow, it would be necessary create a server environment for my application instead of shipping everything through the web client as was previously planned.
The Implicit Grant Flow does not require a server however. In this flow, the entire exchange happens client-side. This simplifies implementation but it does come with a catch. No refresh token is provided. What this means is that tokens received from this flow are short lived, and once expired, need to be fetched again by repeating the entire flow. In the Authorization flow, only step 4 in Fig. 1 needs to be repeated to renew an expired token.

Longer lasting tokens was the primary reason the Authorization flow was chosen and implemented for authenticating party-hosts. Firebase provides a service called Cloud Functions, which makes it fairly simple to upload modules to a managed server environment [Firebase authors, 2018]. It saved me the burden of configuring and deploying my own environment to something like Amazon Web Services.

2.2 Room Creation

To allow party-goers to become part of a room and start adding songs, Qspot needed a way to uniquely match a party-goer to a party-host’s room. The host would need some degree of control over who is added, but this would have to be weighed against the amount of frustration experienced in joining the room. If this frustration is too great, a party-goer might be discouraged from using the application. Conversely, if it is too difficult for the host to add a lot of people to a room quickly, they may also be discouraged or unhappy. There are several approaches that were considered.

An invite system is extremely popular in applications with the concept of a group or room. The owner or host of the room would send out invites via email to guests who then click a link in the email and are granted access to the room. A friend system makes this process
even faster. With a known list of friends, the owner can instantly add anyone amongst them. The problem with this approach is the aforementioned frustration that would be encountered. First time party-goers need to be added quickly. Qspot does not have the friend infrastructure of something like Facebook. Taking the time to make sure every attendee of the party has added the host as their friend, or given the host their email, is impractical. Many people attending the party could even be strangers to the host.

Another possibility was some sort of location based system. Many companies are experimenting with near-field location based grouping with devices like bluetooth beacons [Reddy, 2014], or less accurately with plain old GPS. This could make a lot of sense and addresses the problems with a friend system. However, these location methods have their disadvantages too. Requiring hosts to buy a bluetooth beacon to host their Qspot party is out of the question. The cost and lack of convenience is too large a barrier for the average person. GPS is not helpful if the locations your application needs to consider can be as specific as an apartment. You do not want to give the entire apartment building access to the party for example.

The inspiration for the approach that was finally chosen came from a video-game company called Jackbox Games. They create multiplayer games where every player uses their smartphone to join a room and play a game together. Evidently their problems match Qspot’s quite closely in this regard. Their solution was to use an easily identifiable code that corresponded to a unique game room. The code is displayed on a TV, and players then enter the code into their phones to join [Jackbox authors, 2018]. This strikes exactly the right balance between accuracy and frustration that Qspot needs.
The final implementation uses a four letter code that is randomly generated for each party. To ensure uniqueness and protect against race conditions, a transaction based operation makes sure collisions (a situation where an already occupied code is randomly generated again) are successfully resolved. If two parties generate the code “TIOS” for example, whoever was first to generate gets “TIOS,” while the other party gets “TIOS1.” The third would get “TIOS2,” and so and so forth increasing the number postfixed to the code.

2.3 Searching For Songs

Once inside a party, a party-goer needs to be able to search for tracks they want to add to the queue by entering in a query indicating the name of the song, artist, or album. There were relatively few approaches to consider for this component, and not a lot of freedom involved in implementing it. Since songs on Qspot are played through Spotify, only songs from Spotify’s API are compatible [Spotify authors, 2018].

Still, the mechanics of a good API search interface can be difficult to get right, and some key choices can be made around implementation that really change the experience of a search. One such option was whether to choose a live search versus a static search. In a live search, results come in and are displayed as the user types their query. No extra input is needed. In a static search, the query does not launch until another specific command is given, often by pressing the “Enter” key. The live search is harder to implement, and pulls more data over the network on average since some queries could be launched unnecessarily. However, they are often more attractive to users than a static search. Seeing the results roll in as you type has a certain premium feel to it. The Spotify search API
returns a relatively small amount of data too, so ultimately a live search approach was chosen [Spotify authors, 2018].

Making an effective live search relies on debouncing [Zuberi, 2018]. In simple terms, this takes an action that would have happened multiple times and collapses it down to one action. The “action” in this scenario is the user typing a key in the search box. A bad live search would be one that makes an API request on every keystroke. This wastes data and wastes processing power rendering search results when the user probably has not finished typing yet. If we debounce this action, we can specify a time frame where we wait to make sure no new keystrokes have occurred before performing the API request.

This behavior is observable in the search page of the completed application. Try typing a search query at a reasonable speed and you will notice the results won’t appear until you have finished typing.

2.4 Music Playback

Qspot is a music playing app so it would not do much good if it could not play music. Spotify’s Web Playback SDK is the crucial component of Qspot that actually allows it to stream music from Spotify [Spotify authors, 2018]. It creates an encrypted player within the browser and responds to commands like “pause,” “play,” and “skip.” It was released late last year and is still in beta. Unfortunately, due to a number of bugs and shortcomings with its functionality, trying to make it work for Qspot was very difficult.
Qspot controls a queue that only itself knows the structure of. In Spotify’s own desktop application, a queue of upcoming songs exists that you can manage to control what songs are coming up next, but when working with the Playback SDK, there is no way to manage a queue. Users of the SDK can request a list of songs to play that will then be queued up by the SDK, but as soon as another list is requested, the old list is discarded and the currently playing song is replaced. This is very unhelpful if the goal is to get Spotify to understand the structure of the queue. There were two possible approaches that emerged from this limitation.

One approach was to create a Spotify playlist on the host’s account that would represent the queue. You can’t manage the player’s queue through the same mechanism Spotify has access to, but you can manipulate the position of any song in a playlist through their APIs [Spotify authors, 2018]. It would be possible then, to make on-the-fly additions, removals, and position swaps to this playlist to make it match Qspot’s own queue. The downside is you need to rely on the host to start the first song in the playlist through the Spotify’s own application, and not touch any of Spotify’s controls afterwards. If the host plays a different song in the playlist for example, Qspot’s entire understanding of the current state of the queue would crumble. Also, cluttering the host’s Spotify with playlists like these is a very undesirable side-effect.

What Qspot really needed is a way to be signaled when a song being played by the playback SDK ended, or was skipped by the host. Then it would have no need to worry about Spotify’s understanding of the queue. Qspot already knows what song is up next. If it receives a signal that the current song is over, it can issue a request to play the next one. But unfortunately the SDK does not provide any specific signals that fire when one of these events occurs. The only information provided is an occasional update that let’s the user of
the SDK know what state the player is in. This includes values like the current position of
the song, whether it is paused or unpause}\,d, and what the names of the current track,
previous tracks, and next tracks are. By carefully interpreting the information in these
updates, it was possible to reliably find out when a track had ended or had been skipped.
This is what enables Qspot to play through the queue properly in the final implementation.

2.5 Queue Ordering

From the beginning I knew I wanted to have some form of intelligent ordering to the songs
in the queue. Handling additions to the queue on a first-come-first-served basis would be an
extremely problematic ordering for a number of reasons. Anyone could spam the queue and
fill it with their own songs before anyone else had a chance to add anything. There could
have been some system used to mitigate spamming say with a timeout on adding songs;
only one song every 5 minutes for example. However, this would need to be tediously
adjusted based on the number of people submitting songs and how frequently everyone is
submitting to not frustrate users. It does not solve the core problem: first-come-first-served
creates an unfair ordering.

The intelligent method that was initially thought of was a round-robin style of ordering. A
very fair way to represent a group’s tastes as that group listens to music together is to play
one song chosen by each person one after the other until you have played a song from
everyone in the group, then repeat this process. The only potential caveat is that in a
system like Qspot, you cannot wait on certain people to submit songs at certain times. You
have to able to handle their additions as they come in, and order them on the fly. Some
people in the group might not submit a song at all before several rounds have gone by.
The eureka moment came when I realized that for a given song, there were only two numbers necessary to create a perfect ordering: the time that song was added to the queue (we'll call this $T$), and how many songs the user that submitted this song had submitted previously (i.e. if this song is the $I$'th song submitted by that user, the important number here is $I$). Retrieving the proper order of the queue then becomes as simple as sorting the songs first by $I$ and then by $T$. Everyone’s first song comes before everyone’s second song and so on. To decide which “first song” comes first, the second sort is performed, leaving it down to time (i.e. whoever added their first song before everyone else’s first song gets to be first in line).

### 3.0 Results

All of the objectives outlined for Qspot were accomplished. The app is being hosted live through Firebase’s static hosting service [Firebase authors, 2018]. There are some quality of life aspects and ending portions of the user flow that did not make it into the final application. I had planned to create a more visually striking and user-friendly interface. Priorities took over and forced me to keep my UI minimal. Only basic HTML elements are used, and just enough styling work was done to make it usable on mobile devices.
Fig. 1 is a screenshot of what I would consider to be the main view of the application. Nothing is omitted here. This is the entire user interface. Under “Now Playing,” the name of the song currently playing through the host’s Spotify account is shown. The “Play” button reflects the paused/resumed state of the song. The “Start Party” button, only relevant to hosts, will begin playback of the queue. The numbered songs 1-6 are the songs currently in the queue. The picture just below “Qspot” is the profile picture associated with the currently logged in user’s Spotify account. The picture can be clicked to log out.
The Search page shown in Fig. 3 has a single input box to type in the user’s search query. The results of the search render below dynamically as you type. The “Add track” button next to each song will add the corresponding song to the queue.

The Join page shown in Fig. 4 allows one or two functions depending on if you are logged in with a premium Spotify account. All users can enter a four letter code in the input labelled “Enter party code.” Once they do so and hit the “Join” button, their information is added to the party matched by that code (if one exists), making them a member of the party. “Create
Party,” is only relevant to premium Spotify accounts. Once clicked, a party is generated with it’s own unique code and with that user assigned as the host.

Welcome Back

The Login page pictured in Fig. 5 is the most minimal of all. It provides two button to start the login flows of each type of account: Google, and Spotify.

Probably the two most unfortunate omissions are features which handle the tail-end of the main user-flow. Once a party-goer is done with a party, they should be able to leave that party to allow them to join a new one at a later time. Once a party-host is done with a party, they should be able to “end” that party. This operation would delete all tracks and members of the party and inform each member that the party has concluded. Both of these were not achieved in time for submission.

There also does not exist a way to view or share a host’s party code once it is generated. For now, I rely on my ability to view the database to check what code was generated.

Everything that was implemented is functioning properly as far as my testing is concerned. There were some limitations and bugs with the Web Playback SDK that are out of my control. The SDK is only supported on the desktop version of Chrome, Firefox, IE11, and
Edge. There is no support for desktop Safari or mobile browsers of any kind. I also identified and filed a bug with the SDK that caused it to not fire events for important state updates if those state updates appeared identical to the previous update. For example, a state update that lets my application know a track (we’ll call it \textit{Track1} ) has started playing will have it’s “position” property set to 0. If you let \textit{Track1} play for any given length of time, and then manually seek to the beginning of the track before another state update has fired, the application \textit{should} receive a new state update to let it know what’s going on. However, this does not happen since that hypothetical state would still have a “position” of 0. The SDK does not recognize this as a “changed” state, and therefore does not fire an event for it.

There also exists a potential problem when the application is running at a larger scale that I was not able to test. Spotify applies a rate limit to it’s API [Spotify authors, 2018]. This means that per application registered to use Spotify’s API, a limit is placed on the amount of requests that application can make to a particular endpoint within a certain period of time. Potentially, this means that if a large number of users were to load track information for the queue all at once, the rate limit could kick in and fail some of these requests. There is further work that could be done to mitigate this by grouping separate requests for track information into a single request.

The most important issue that I felt was dealt with in my project was interpreting state updates from the Web Playback SDK to detect important events like a track ending or being skipped. Since the SDK’s release, a lot of people have been asking for an easy way to detect these events with no response from Spotify [Laarhoven, 2018a] [Klein, 2017b]. My work, if provided as a package/tool or as an example, could help a lot of people.
Generating unique, short codes and guaranteeing their uniqueness whilst protecting against various race conditions was an important accomplishment. There still does not exist a lot of good information around on how to accomplish this in a real-time document system like Firestore’s. This technique is applicable to a lot of different use cases across various applications.

With more polish I’m confident my application could foster a community of real world users. I think everyone can emphasize with the feeling of wanting to listen to your own song at a party. There are millions of Spotify premium users with compatible devices that could take advantage of this application to enhance the social quality of their parties. I was very attracted to the idea of making an application that could not only accomplish this, but do it in a way that does not try too hard to hold the attention of the user. Users should pick a song, put the app away, and get back to enjoying their own or their friend’s music together. At the very least, I feel as though I’ve prototyped something that makes this a reality.

4.0 Conclusion

Constructing this application felt a lot like doing 10 different smaller projects at once. There are so many moving pieces to do what in the end appears as a very simple result (no thanks of course to the exceedingly minimal user interface). In my rough estimate, the project contains around 2500 unique lines of code. Some of those lines felt trivial to write, some felt impossible to. And like any reasonably complex web application in the modern world, it would not have been possible to complete without standing on the shoulders of giants that make powerful libraries like Firestore. I hope to pour much more work into Qspot
going forward. I want to see it become an end-to-end production ready application. I want to see it look a bit prettier too.
References

TouchTunes authors (2018) Overview, TouchTunes Music Corporation
[http://www.touchtunes.com/about/overview/]

Firebase authors (2018) Authenticate Using Google Sign-In with JavaScript, Firebase
[https://firebase.google.com/docs/auth/web/google-signin]

[https://developer.spotify.com/documentation/general/guides/authorization-guide/]

Firebase authors (2018) Cloud Functions for Firebase, Firebase
[https://firebase.google.com/docs/functions/]

Trips Reddy (2014) 15 Companies From Airports to Retail Already Using Beacon Technology, Umbel
[https://www.umbel.com/blog/mobile/15-companies-using-beacon-technology/]

Jackbox authors (2018) How to Play Jackbox Games, Jackbox Games
[https://jackboxgames.com/how-to-play/]

Spotify Authors (2018) Start/Resume a User's Playback, Spotify for Developers
[https://developer.spotify.com/documentation/web-api/reference/player/start-a-users-playback/]

Spotify Authors (2018) Search for an Item, Spotify for Developers
[https://developer.spotify.com/documentation/web-api/reference/search/search/]
[https://medium.com/walkme-engineering/debounce-and-throttle-in-real-life-scenarios-1cc7e2e38c68]

Spotify Authors (2018) Web Playback SDK Quick Start, Spotify for Developers
[https://developer.spotify.com/documentation/web-playback-sdk/quick-start/#introduction]

Spotify Authors (2018) Playlists, Spotify for Developers
[https://developer.spotify.com/documentation/web-api/reference/playlists/]

Firebase authors (2018) Firebase Hosting, Firebase
[https://firebase.google.com/docs/hosting/]

Spotify Authors (2018) Web API, Spotify for Developers
[https://developer.spotify.com/documentation/web-api/]

Rick van Laarhoven (2018) Detect the end of a track, Github
[https://github.com/spotify/web-playback-sdk/issues/35]

Klein (2018) Detect end of a track’s playback with Spotify Web API / Playback SDK, StackOverflow