Progress Report

Harry Verhoef, u1706021

November 25, 2019

Contents

Chapter 1

Introduction

Too often in any situation where multiple people are listening to music from the same source, there is only one person/device controlling what music is being played. As a result, many people are subject to music they are not particularly fond of. This project aims to give everyone in this situation a voice in the form of a vote that can be cast suggesting the next song to be played from a selected playlist. The song with the most votes will be en queued at the end of the current song. The idea is to introduce the most democratic music environment possible, one where the majority of people listening are happy with what they're listening to. With this in mind, users will also have the ability to up-vote and down-vote a current song. On top of that, a machine-learning module will be built that considers:

- Data from the more favoured (most up-voted) songs, (genre, artist, tempo, "danceability", etc.)
- Personal preference data from the users' Spotify account.
- User credibility (vote-weight)
- Situation setting (party, gym, radio, etc.)

And recommends the next songs to be put up for voting. Therefore, unlike any other AI song recommendation system, this one will take into account the multiple users listening to the music and their preferences.

Chapter 2

Background Research

There have been previous attempts at making music playback a democratic process: Spotify have introduced a social listening system [1] that lets users join a party and from there any user has complete control over the music playback, which is a system that can be exploited by any user. Also only works for users who have the Spotify app. This project will work with only one user in a lobby required to have the Spotify app.

An app called "Outloud" [2] provides a similar service to that outlined in the introduction, it lets users vote for the song to be played, but does not use this data to build a neural recommendation system that suggests the music most likely to please everyone in the lobby, it only uses the prebuilt shuffle tie-breaker, which only tailors to the preference of the host user.

The react-native online docs [3] were used to create sample test applications before any start was made on the project application. The applications built were very simple and did not focus on any integration at all.

Some background research has been made in to the usage of the react framework, since this is a new approach to front-end development to the developer. The documentation used for research purposes was the react.js docs [4].

Chapter 3

Analysis of problems and issues

As discussed in the introduction, the main problem being addressed is the idea of having an anti-democratic music environment within which a minority of those listening are satisfied with the music.

This project is of course not the only way this issue could be addressed. For example, a module could be written that allows multiple users to connect to one device over bluetooth and play music from whichever is attempting to play. On one hand, this solution is straightforward to comprehend with regard to the solution this project proposes. On the other hand, it may be considerably harder to implement - having to deal with low-level network interactions conforming to Bluetooth Special Interest Group (SIG) standards [5]. This solution would also only work where a bluetooth device is playing the music. Moreover, this solution is more anarchic than it is democratic, providing an environment with no rules or governance, any user can take over.

There is an abundance of challenges involved in this project, namely:

- Learning Objective-C and then creating an Objective-C bridging module that provides easy integration between the Spotify SDK and React Native.
- Learning and developing a React-Native UI that is both function, responsive and aesthetic.
- Learning Node.js and then writing a large Node.js back-end API that acts as an intermediary between the client an any back-end services such as the recommendation system or Spotify Accounts Service.

- Implementing the O-Auth based authorisation system for Spotify accounts.
- Researching, designing and implementing the recommendation system.

Chapter 4

Design

To begin with, a "sitemap" was designed to provide a simple overview of the structure of the client-facing side of the application. Each of these "pages" represent a JavaScript file.

To begin with, a "sitemap" was designed to provide a simple overview of the structure of the client-facing side of the application. Each of these "pages" represent a JavaScript file, which in turn represents a react component. This is represented by Figure 4.1.

Since the process of choosing the next track to play is fundamental to the purpose of the application and what makes it stand out from the aforementioned similar solutions in section 2, a flowchart representing this mechanism was designed (Figure 4.2). The inner workings of the recommendation system have not yet been designed and therefore cannot be expanded upon. Once research has been conducted into the neural network that acts as the recommendation system, a separate flowchart specifically for the recommendation system will be built.

Figure 4.3 is an abstract chronological representation of the Spotify account authorisation that will take place within the application.

1. Application switch for Spotify Authorisation.
2. Return to project application with authorisation (accept/reject).
3. POST request to /swap path, with an authorisation code that can be exchanged for an access token in the body of the request.

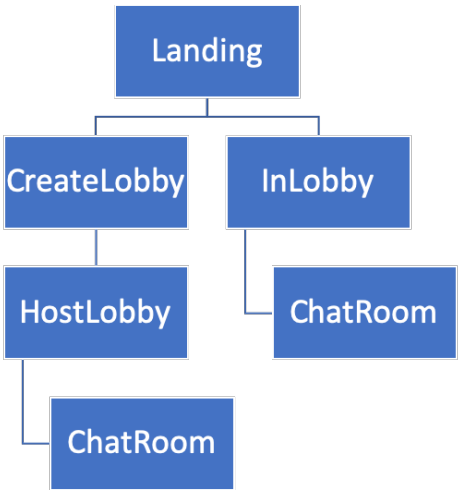

Figure 4.1: Sitemap

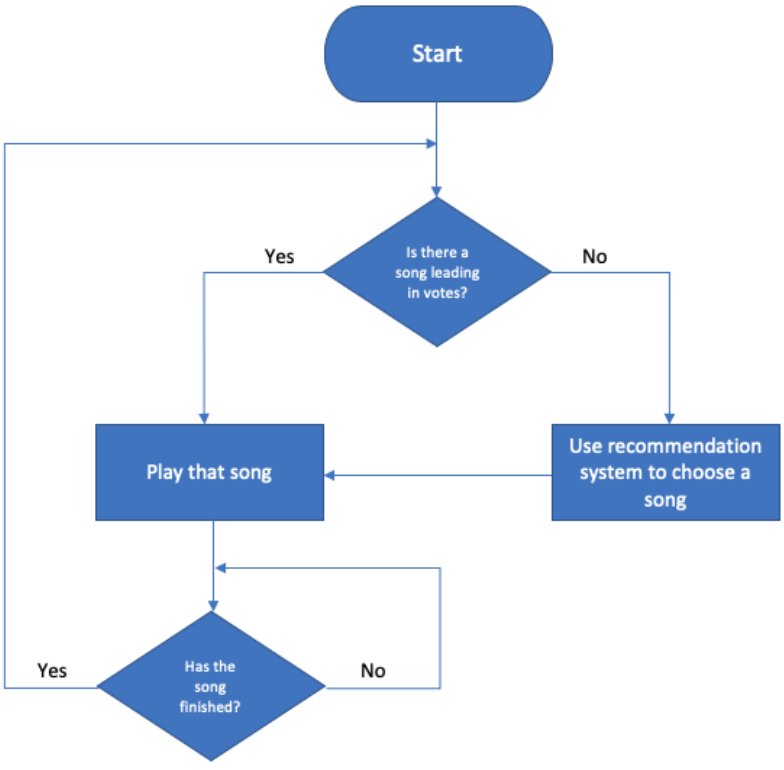

Figure 4.2: Enqueue mechanism flowchart

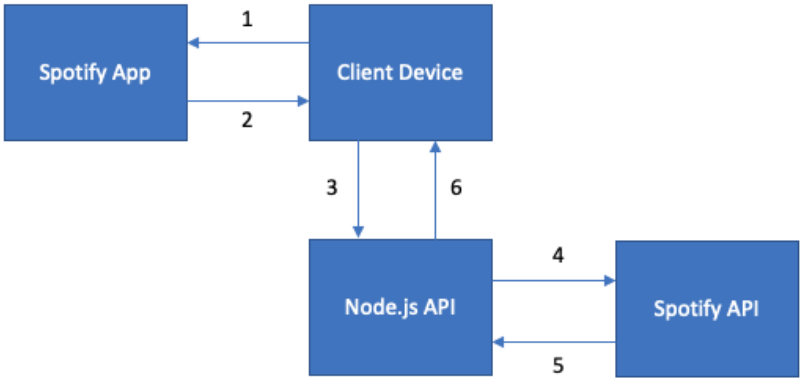

Figure 4.3: Spotify account authorisation

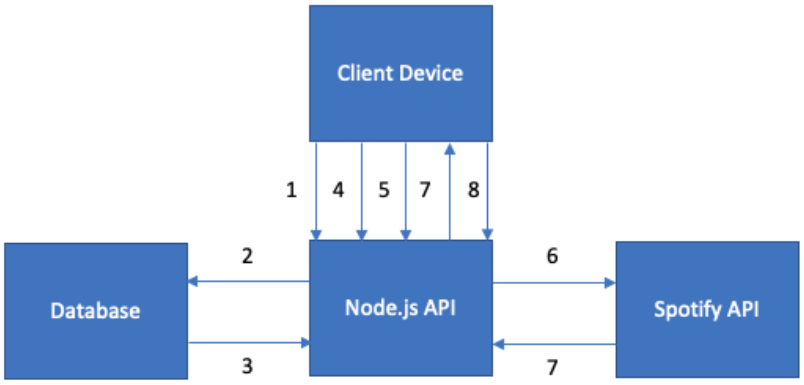

Figure 4.4: Playlist creation

4. POST request to Spotify web API /token endpoint, containing this authorisation code, API keys and the required scope of the application (permissions granted by the user).

5. Response from Spotify web API, crucially containing the access_token and refresh_token for the given user. At this point the Node.js API will store this in the user object (and database) so that these tokens can be used in the future at any point.

6. Response from Node.js API, specifying the relevant tokens to start a Spotify session through the bridging module.

Figure 4.4 is also an abstract chronological representation, this time of the lobby-creation procedure.

1. User opens the project application and a WebSocket connection is established with the Node.js API, sending the unique device id.

2. Send request to obtain user info based on unique device id from database.

3. Database responds with the user information, such as vote-weight, which is then placed into the user object. If there is no record of the unique device id, then a new one will be made and the corresponding user object will be assigned the default user values.

4. User presses the "Create new lobby" button, and is "redirected" to the CreateLobby "page".

5. User completes the aforementioned (and hence omitted) authorisation procedure, then presses the "select playlist" button.

6. Node.js requests playlist data from Spotify web API, using the playlists endpoint. Where the user_id is also retrieved from the user object.

7. Playlist objects are sent in encoded JSON form in a HTTP response body, which are not stored by the Node.js API.

8. Playlist objects are returned to client device, and are displayed so that the user can choose which base playlist to use for the lobby.

Chapter 5

Choice of methods and tools

Since the majority of technologies being used in this project require research, such as react-native or machine-learning, the development methodology of choice is agile. This is because a dynamic specification is required so that upon further research, certain requirements can be amended suitably.

React-Native is being used because it is a new JavaScript framework, one that is exciting and has lots of potential. It can be exported to both Android and iOS devices [6] so that in the future, a greater audience can use the application. For now, the development is focusing solely on iOS. Since the application is being developed on a mac, and tested on an iPhone, it is a much more compatible operating system to build the application with.

When considering the versatility of the proposed application, the idea that it can be used in a variety of situations, such as a car or a gym or a radio, means that the number of users within a lobby also varies heavily. With this, it is important to make sure that the back-end application is capable of handling lots of user connections. Therefore, the backend API is being developed in node.js, since "it's capable of handling a huge number of simultaneous connections with high throughput, which equates to high scalability" [7]. Not only this, but with node.js comes npm, the node package manager. Using npm, packages such as express, axios or socket.io, are incredibly easy to install and use.

Objective-C is an old language and one that has been replaced by a more elegant language: Swift. However, there is more documentation on using native modules in Objective-C, which is essential to the application. Therefore, Objective-C is the chosen language of the native bridging module.

The database being used for this project will be MongoDB. This is again

due to it's high scalability [8], but also due to it's schema-less property of being very able to work with heterogeneous data since the typing is much less strict. This is important for this project since the application in future will be developed for both Android and iOS, two very different operating systems. For example, Android devices have a different representation of unique id to iOS devices. Therefore, a schema-less database would be more suitable for users who are identified using id's that do not follow the same pattern.

The neural network recommendation system, along with the back-end API and database, will all be hosted by Amazon Web Services (AWS), since it is a cheap, fast, durable and secure platform [9]. Apple introduced a security feature called App Transport Security in January 2017 [10], meaning that in order for an app to be published on the app store, it must use no HTTP connections, only HTTPS. Using AWS Certificate Manager, it is easy to request an SSL/TLS certificate and therefore use HTTPS across the project [11].

Chapter 6

Work completed

The Objective-C bridging module has been developed completely, with most methods written in the AppDelegate.m, including the ones specified in the Spotify iOS SDK Tutorial [12]. Then in the bridging class, these methods are called within an exported method, which uses the value of the call and passes it into the callback. Figures 6.1, 6.2 and 6.3 are a demonstration of this infrastructure - It is used for all native module methods.

This infrastructure follows for all of the 5 other implemented exported methods so far: invokeAuthModal (trigger the app switch and begin the authorisation phase), getPlaylists (gets an NSMutableArray of STPAppRemoteContentItems (playlists)), playURI (Takes a String representation of the song URI and plays that song), connectAppRemote (attempts to connect the app remote), queue (Takes a string URI and queues the song). The bridging module contains many more methods that aren't exported to the react-native end, they're just used "internally", mostly for authorisation and handling the Spotify session.

With regard to the node.js backend, both the user and lobby objects have been created, and using them, the functionality to create/join a lobby, authorise a Spotify account and upload a playlist, is completely implemented.

It is worth noting, that as shown in Figure 6.4, the relation between the lobby and the user is an aggregate one. In that if a lobby ceases to exist, it does not therefore imply that the users that make up the lobby are then deleted too. This is the preferred relationship because a global list of users is kept on the node.js API, from which their lobby cannot be derived without using their getLobby() accessor method.

The Node.js API uses a combination of both WebSocket and RESTful

```
- (BOOL)skipSong {
  if (self.appRemote.isConnected) {
    NSLog(@"Attempting to skip song and appRemote is connected");
    __block dispatch_semaphore_t skipSema = dispatch_semaphore_create(0);
    __block BOOL success = NO;
    [self.appRemote.playerAPI skipToNext:^(id  _Nullable result, NSError
        * _Nullable error) {
      if (error) {
        NSLog(@"Error skipping next song: %@",
            error.localizedDescription);
      } else {
        NSLog(@"Skipped song");
        success = YES;
      }
      dispatch_semaphore_signal(skipSema);
    }];
    // dispatch_time takes a dispatch_time and a delta (measured in
        nanoseconds)
    dispatch_semaphore_wait(skipSema,
        dispatch_time(DISPATCH_TIME_NOW,400000000));

    return success;

  } else {
    NSLog(@"Attempting to skip song and appRemote is not connected");
    return NO;
  }
}
```

Figure 6.1: Objective-C skipSong() method writtin in AppDelegate.m.

```
RCT_EXPORT_METHOD(skip:(RCTResponseSenderBlock)jsCallback) {
  NSNumber *result = [NSNumber numberWithBool: [self.appDelegate
      skipSong]];
  jsCallback(@[[NSNull null], result]);
}
```

Figure 6.2: Objective-C skip method being exported to react-native as an attribute of the NativeModules object.

```
spotifySDKBridge.skip((error, result) => {
    if (result) {
        Alert.alert("Song successfully skipped");
    } else {
        Alert.alert("Song could not be skipped");
    }
});
```

Figure 6.3: React-Native skip method being invoked on the bridging module instance.

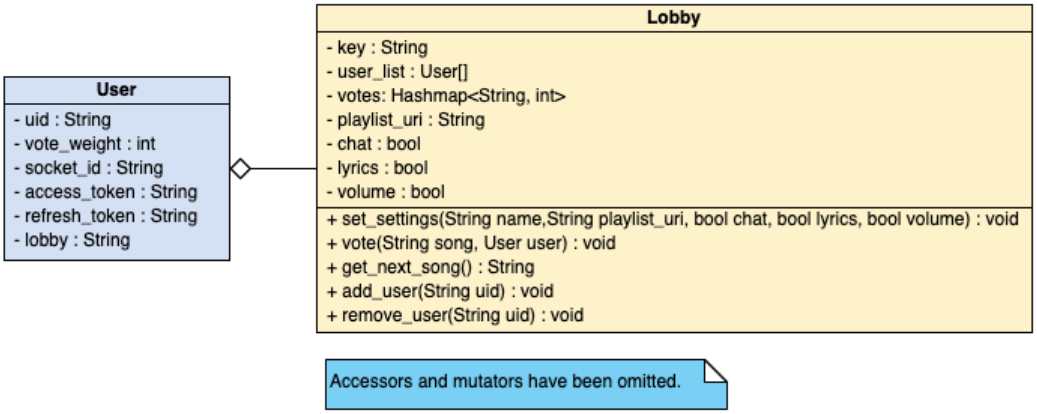

Figure 6.4: Object Model Diagram for User and Lobby.

API paths. The WebSocket part of the API is useful for when there's a need for stateful full-duplex communication such as the chat room or constantly getting the state of the song playing in the lobby. The RESTful side of the API is used for communication that is more occasional, such as when a user needs to authenticate their Spotify account, or get the next recommended songs.

As for the Graphical User Interface (GUI), both Landing.js and CreateLobby.js have been developed to a sufficient level for full testing. The GUI is something that is not deemed overly important in the specification, it will be taken more seriously towards the end of the development when most of the functionality described in the specification is fully developed. Navigation between pages is handled by react-navigation [13] which models the navigation of a user in the form of a stack. Considering the sitemap 4.1, a user can progress down the depth of a sitemap and then go "back" using the gesture of a swipe [14]. This stack has to be defined and exported to index.js, then the navigate object is passed in to each component (each page), from which the navigation and user-story can be controlled further.

The carousel of playlists demonstrated in Figure 6.6 has been implemented using the react-native-snap-carousel package [15].

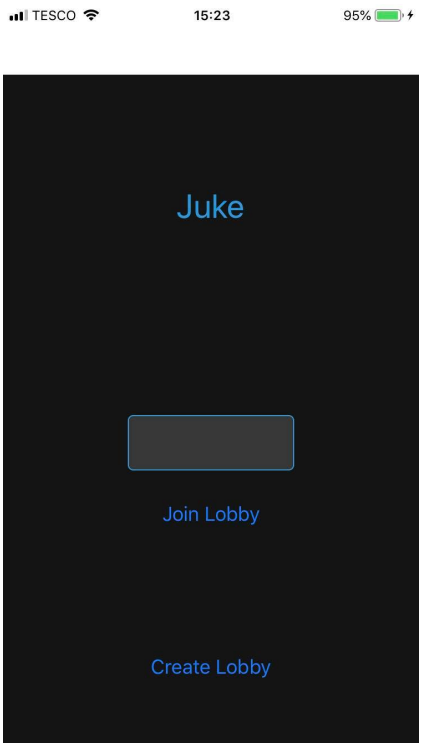

Figure 6.5: Landing.js

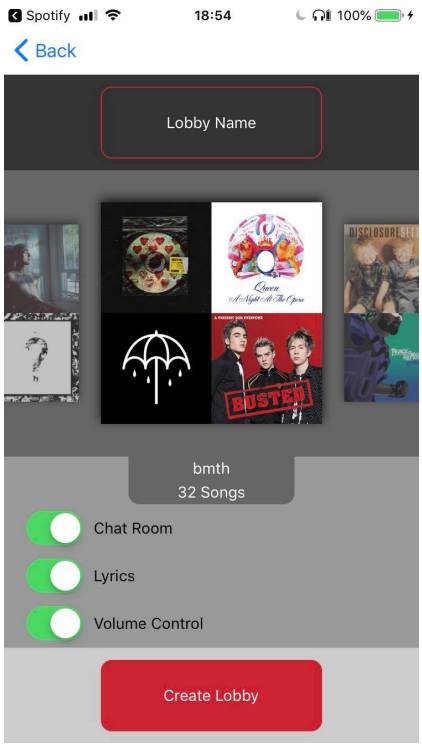

Figure 6.6: CreateLobby.js

Chapter 7

Progress issues

When writing an application using react-native, and testing it on a physical device, all you have to do is connect your apple device to a mac, open xcode and build from the xcworkspace located in the ios folder which is created after initialising the react-native project. After approximately 10 minutes the react-native app is built to the device and is automatically run. A JavaScript metro bundler server will be initialised on the mac from where the device will download the JavaScript bundle. Once the react-native code is changed, requesting the JavaScript bundle once again from the bundler service will provide an up-to-date app. Therefore, developing the UI for a react-native application is much like developing a UI for a website, you save the changes and refresh. However, with this project it is necessary to develop a native module that integrates between the Spotify SDK and the react-native end. Such a module requires that once changes are made, the entire app be re-built to the device. As a result, it took approximately 10 minutes to do every single unit test. This made the development of the module incredibly inefficient.

Authorisation with Spotify Accounts Service is a critical process in the user story of the host - In order to play any music from a Spotify account, that account must first be verified by Spotify themselves. Figure 7.1 is a diagram of how such authorisation takes place.

The first step as shown in Figure 7.1 was implemented in the Objective-C bridging module, which triggers an app switch to Spotify and prompts the user, asking for their consent to let the project access their Spotify data. Once confirmed or denied, the user is switched back to the project app and a HTTP POST request is sent to the Node.js API /swap path (specified in

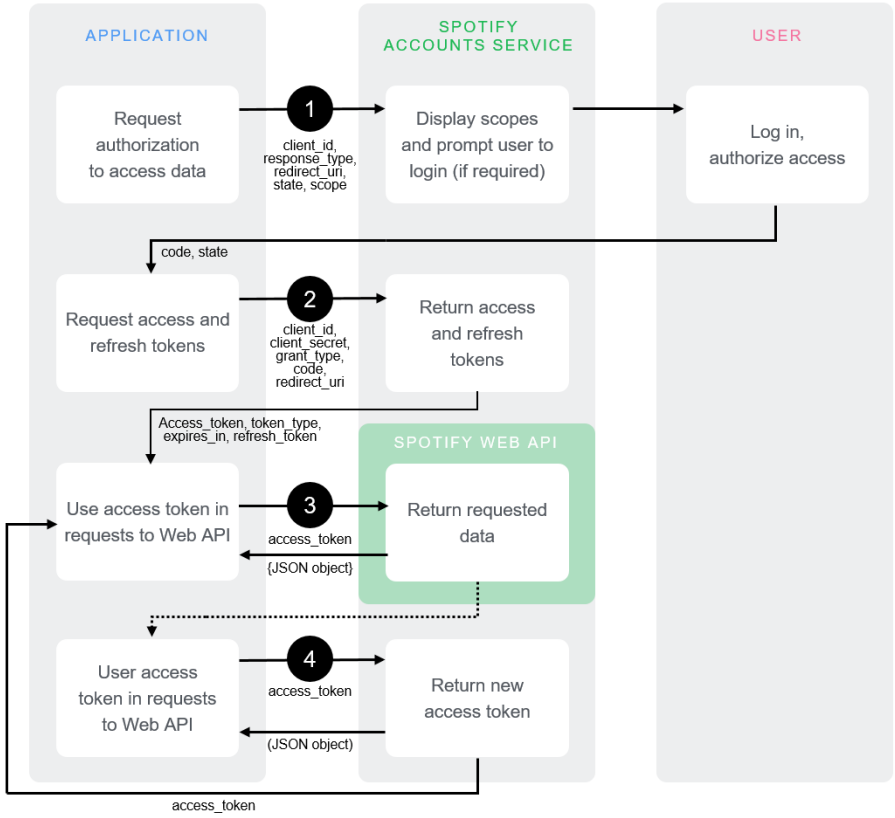

Figure 7.1: Spotify authorisation flow [16]

the Objective-C bridging module), this takes the "code" and "state" from the response body, then executes step 2. The response from this is sent back to the client device. At this point the access_token can be used to access user data, a native Spotify session can be started, and thus the app remote can connect. With the app remote connected, music playback can be controlled.

This took a very long time to develop, partially due to the compiling times outlined earlier, but also due to seemingly ambiguous semantic errors. For example, during the app switch, the Spotify app would crash, this was because for iOS, in order to initialise a Spotify session (required for authorisation), Spotify must first be "woken up" by playing the user's most recently played song. Or, it could crash if some of the exported methods aren't dispatched to the main thread when called.

In react-native, native modules are exported and along with them, an optional collection of exported void methods. Since they must be void, you cannot use the result of their computation without passing in a callback as a parameter. During the authorisation process, 2 methods are invoked on the bridging module, instantiateBridge and auth. Both of these methods have only one parameter, an ES6 arrow function callback, which is converted to a RCTResponseSenderBlock using the RCTConvert class. The parameters of each callback are an error and result object, as per the react-native documentation [17]. However, due to the nature of the Objective-C compiler being used, during the development of the authorisation procedure, the values of these objects seemed to be independent of the results of the invocation. For example, the user could deny the project access to their data, yet the result of the auth call would suggest successful authorisation. Or, the client device could be without the Spotify app, yet apparently the bridge with the Spotify SDK was instantiated successfully. Often, the callback was invoked before the exported method had finished executing. This behaviour defeats the point of callbacks, making controlling the authorisation flow very difficult. The problem was that the aforementioned Objective-C compiler had methods return any constant expression before it had finished executing the method body. This obscure problem was solved through the use of semaphores, before an Objective-C method would return any non-constant value, it must first signal a semaphore. Of course no semaphores have been used on methods that may be dispatched to the main thread.

As part of the lobby creation process, users must select one of the playlists they either own or follow to be used as their base playlist. In react-native, images must be rendered with a compile-time constant path. For this project,

it is impossible to know the path of the user's playlists at compile-time. Using a slight loophole, the source of an image can be set to a constant URI to the back-end API's /get-image path, with an editable body. Thus, the temporary source of the playlist icon can be set as a parameter in the HTTP body, and upon receiving the request the back-end API will request the temporary image and pipe it back to the client device. Here, the API is being used as a proxy in-between the client and the temporary image so that image components can be dynamically rendered.

Due to all of these issues, an updated timetable, Figure 7.2 has been created. This timetable has extended task 6, the implementation of the ability to join/leave a lobby and vote for their preferred song. At the moment, users can join and leave lobbies, they cannot currently vote for their preferred song at then moment. To account for this increased development time, the new timetable has omitted the need to integrate the application with the Android SDK, since it is not a top priority and the application's full functionality can be demonstrated without it.

Task No.	Task	Time taken	Date (Starting 05-Aug-19)	Dependencies
1	Setting up development environment.	1 week	12-Aug-19	
2	Creating temporary UI and researching react-native.	2 weeks	26-Aug-19	1
3	Integrating project with iOS Spotify SDK.	8 weeks	21-Oct-19	2
4	Designing and implementing a responsive and functional UI for beginning of user-story.	2 weeks	04-Nov-19	1
5	Implementing the ability for users to create and terminate a lobby.	1 week	11-Nov-19	4
6	Implement the ability to join/leave a lobby and vote for their preferred song.	3 weeks	02-Dec-19	2,4
7	Implement the switching of songs to the most voted.	0.5 weeks	05-Dec-19	6
8	Write up the progress report.	0.5 weeks	25-Nov-19	
9	Deploy application on AWS	1 week	12-Dec-19	
11	Implement a vote-weighting system.	1.5 weeks	29-Dec-19	10
12	Integrate with Genius API (for displayable lyrics)	2.5 weeks (Factoring in Christmas holidays)	15-Jan-19	4
13	Implement the displayable lyrics during songs.	2 weeks	29-Jan-20	12
14	Researching neural networks.	1 week	05-Feb-20	
15	Designing the neural network.	1 week	12-Feb-20	14
16	Implementing the neural network recommendation system.	2.5 weeks	29-Feb-20	15
17	Implementing the chat room.	0.5 weeks	03-Mar-20	4
18	Creating and running the test suite.	1 week	10-Mar-20	
19	Preparing the oral presentation.	1 week	17-Mar-20	18
20	Majority of final report.	5 weeks	20-Apr-20	18
21	Finishing touches on final report.	1 week	27-Apr-20	20

Figure 7.2: Updated timetable

Bibliography

[1] "Spotify social listening." https://www.highsnobiety.com/p/spotify-social-listening-feature/. Accessed: 24/10/2019.

[2] "Outloud application." https://outloud.dj/. Accessed: 24/10/2019.

[3] "React-native docs." https://facebook.github.io/react-native/docs/getting-started. Accessed: 24/10/2019.

[4] "React docs." https://reactjs.org/docs/getting-started.html. Accessed: 24/10/2019.

[5] "Bluetooth sig." https://www.bluetooth.com/specifications/. Accessed: 24/10/2019.

[6] "React-native." https://facebook.github.io/react-native/. Accessed: 24/10/2019.

[7] "Node.js scalability." https://medium.com/the-node-js-collection/why-the-hell-would-you-use-node-js-4b053b94ab8e. Accessed: 24/10/2019.

[8] "Mongodb scalability." https://dzone.com/articles/divide-and-conquer-high-scalability-with-mongodb-t. Accessed: 24/10/2019.

[9] "Consider aws." https://www.netsolutions.com/insights/what-is-amazon-cloud-its-advantages-and-why-should-you-consider-it/. Accessed: 24/10/2019.

[10] "Mandatory https." https://forums.developer.apple.com/thread/48979. Accessed: 24/10/2019.

[11] "Aws certificate manager." https://aws.amazon.com/certificate-manager/. Accessed: 24/10/2019.

[12] "Spotify ios sdk (objective-c)." https://developer.spotify.com/documentation/ios/quick-start/objective-c/. Accessed: 24/10/2019.

[13] "React navigation." https://reactnavigation.org/. Accessed: 24/10/2019.

[14] "React native gesture handler." https://kmagiera.github.io/react-native-gesture-handler/docs/getting-started.html. Accessed: 24/10/2019.

[15] "React native snap carousel." https://github.com/archriss/react-native-snap-carousel. Accessed: 24/10/2019.

[16] "Spotify account authorisation flow." https://developer.spotify.com/documentation/general/guides/authorization-guide/. Accessed: 24/10/2019.

[17] "Native modules." https://facebook.github.io/react-native/docs/native-modules-ios. Accessed: 24/10/2019.

Appendices

Appendix A

Specification

A.1 Problem Statement

Generally, in places such as a gym, a party or a car, the music being played is controlled by one person or device, the "host". This mobile application aims to replace this idea with that of a collaborative democratic music playback environment where each user has the ability to make their voice heard. Users will be able to vote for the next song, rate the current song and request to change the volume. While the song that is voted next is likely to be known by a large portion of the lobby, there will still be users who are unfamiliar with the democratically elected song. Therefore, the app will provide on-screen lyrics in time with playback. With these functionalities it is important to let the users in a lobby communicate, therefore each user will have the ability to enter a chat room. In an attempt to prevent users from abusing their voting powers, a vote-weighting system will be implemented so that users who have their recommendations down-voted by a certain percentage will have their vote worth less across all future lobbies. On the contrary, if their suggestions are perceived by other users to be particularly good, then their vote will become worth more.

In the event that no song has been decided as the next song, the obvious option would be to shuffle the playlist to find a pseudo-randomly generated song. The whole purpose of the app is to provide music playback with a high probability that the majority of users will enjoy. Therefore, the application will make use of a neural network to determine the next song in the queue if the vote results in a draw or there are insufficient votes. The inputs for this

neural network will stem mainly from user's most played tracks (if data is available) and the genres/songs most voted for in the past. Feedback will be provided to the network via the ratings of the songs suggested by the neural recommendation system.

The host device will be the one controlling the lobby settings, who is allowed to join the lobby, etc. Instead of having to manually enter a lobby key, host devices can send a deep-link to join the lobby or generate a QR Code.

A.2 Objectives

1. Host:
   (a) he host user will be able to create a lobby and invite users to this lobby either through a key, a link, or a QR code.
   (b) The host user will be able to alter the following lobby settings before and after creation:
      i. Lobby name
      ii. Base playlist
      iii. Maximum users
      iv. Type of lobby (gym, party, car, radio, etc.)
      v. Chat room (enabled/disabled)
      vi. Volume control (enabled/disabled)
      vii. Lyrics (enabled/disabled)
   (c) The host user will be able to terminate their lobby at any time.
   (d) The host user will be able to override the decisions made by the other lobby members, and the whole lobby will be notified.
   (e) The host user will be able to authenticate their account with Spotify accounts service.
2. User:
   (a) Users will be able to join a lobby if they enter the correct key given it is not full.
   (b) Users will not be able to join a lobby that is full.

(c) Users can only be in 1 lobby at a time.

(d) Users can leave a lobby at any time.

(e) Users can choose to show or hide the lobby chat room if enabled.

(f) Users can choose to show or hide the song lyrics if enabled.

(g) Users can thumbs-up or thumbs-down the current song.

(h) Each user has 1 vote for the next song (however, their vote may be worth more or less than 1).

(i) A user's vote weight depends on the number of times they have received a significant number of thumbs-up or thumbs-down for a suggestion.

(j) A user cannot obtain a vote weight of 0.

(k) If the lobby settings allow it, users can vote to turn the volume up or down.

(l) The user can save any song being played to their Spotify library.

(m) Users on any iOS device can download the app from the app store.

3. Miscellaneous:

(a) The backend API must run efficiently in order to be scalable, and therefore multithreading may need to be implemented.

(b) The app UI will be responsive in that it will function as expected on a wide range of mobile devices and tablets, regardless of screen size or aspect ratio.

(c) The lyrics displayed on screen will be in-sync with the song being played.

(d) Develop and use a test suite to determine the reliability and overall functionality of the application.

(e) The neural network will allow users to get smart recommendations on the next songs to play and will be called as a tie-breaker when two songs have an equal number of votes.

(f) The neural network will use the thumbs-up/thumbs-down data on songs it has suggested as part of its feedback, so that it learns as the app is being used.

(g) The neural network will be hosted on the external application server.

(h) Users will be able to download the app from the app store.

A.3 Methods

The mobile application will be developed through the use of the agile development methodology, where the software is incrementally developed and each release is planned for and then tested against said plan. Releases will be easily monitored using a private Github repository (private so that secret API keys are kept secret) and so that code cannot be copied by anyone. Once the project has been developed to a sufficient extent the repository will be made public so that it is open-source, and the secret API keys will be omitted. Requirements will be gathered throughout the development process using the objectives listed above and during the sprint processes, where a new requirement may become necessary or an old one redundant. With each sprint, a miniature test suite will be developed that tests the new release. This test suite will be developed with the sprint plan in mind. Towards the end of the development life-cycle a large test suite will be developed that will test each requirement with a wide variety of test cases.

I will be writing the front-end code in react-native (a JavaScript framework), however bridges between native modules will be needed in order to communicate with the SDKs on iOS. I will be programming these in Objective-C and Java, respectfully. For the back-end the API will be written in node.js, another JavaScript framework. It is currently unsure as to what language will be used to write the neural network, this will be determined during the neural network research and design phases (task 13 and 14). A database will need to be utilised to store user data. For this, a MongoDB database will be used as a schema-less database to help increase scalability.

A.4 Timetable

Task No.	Task	Time taken	Date (Starting 05-Aug-19)	Dependencies
1	Setting up development environment.	1 week	12-Aug-19	
2	Creating temporary UI and researching react-native.	2 weeks	26-Aug-19	1
3	Integrating project with iOS Spotify SDK.	8 weeks	21-Oct-19	2
4	Designing and implementing a responsive and functional UI for beginning of user-story.	2 weeks	04-Nov-19	1
5	Implementing the ability for users to create and terminate a lobby.	1 week	11-Nov-19	4
6	Implement the ability to join/leave a lobby and vote for their preferred song.	1 week	18-Nov-19	2,4
7	Implement the switching of songs to the most voted.	0.5 weeks	18-Nov-19	6
8	Write up the progress report.	0.5 weeks	25-Nov-19	
9	Implement thumbs-up/thumbs-down on current song.	1 week	01-Dec-19	6
10	Implement a vote-weighting system.	1.5 weeks	11-Dec-19	9
11	Integrate with Genius API (for displayable lyrics)	2.5 weeks (Factoring in Christmas holidays)	28-Dec-19	4
12	Implement the displayable lyrics during songs.	2 weeks	11-Jan-20	11
13	Researching neural networks.	1 week	18-Jan-20	
14	Designing the neural network.	1 week	25-Jan-20	13
15	Implementing the neural network recommendation system.	2.5 weeks	14-Feb-20	14
16	Implementing the chat room.	0.5 weeks	17-Feb-20	4
17	Creating and running the test suite.	1 week	24-Feb-20	
18	Preparing the oral presentation.	1 week	02-Mar-20	17
19	Integrate app with Spotify android SDK	1 week	09-Mar-20	2
20	Majority of final report.	6 weeks	20-Apr-20	17
21	Finishing touches on final report.	1 week	27-Apr-20	20

A.5 Resources

In order to develop and simulate and iOS app, physical iOS devices and a machine with mac OS are required, this will be needed from start to finish. In order to integrate the app with the Spotify android SDK, I will need an android device and Android Studio will need to be installed, in order to simulate android devices of different size and aspect ratio.

In order to learn about and design my neural network I will need access to "S. Rogers and M. Girolami, A first course in Machine Learning, CRC Press, 2011. Ch1", which is available in the CS342 online material webpage.

I may also need to have an application server being run so that the project can access it's own API constantly. This will be acquired through an online

application server hosting service. This will be needed once the project will require testing of multiple users in a lobby while in different networks, and during the development of the neural network.

A.6 Risks

I only have access to one mac OS device and therefore if this one breaks I will not be able to continue iOS development until it is replaced, this would add potentially 2-3 weeks of project delay, depending on the development lifecycle with respect to the project timetable. As a backup strategy I will be able to set up a virtual mac OS emulator on my desktop PC from which I can plug in the iPhone and continue iOS development. If my laptop were to break then all my code would seemingly be gone, thankfully I will be using Github as version control and as a result I will be able to clone the repository to any new machine and continue from my most recent commit.

Another risk is that development is taking too long, in this case I would remove whatever I deem to be the least priority task out of the project timetable. For example, the lowest probability task is probably the chat room, this is additional functionality that, if not fully implemented, will not hinder the overall purpose of the application. This would then free up time that could be used completing a task that has higher priority.

A.7 Legal, professional or ethical considerations

I am aware that Spotify reserve the right to prevent the application from using both their SDK and web API. If this were to be the case then I would implement the project but with another music streaming service instead like amazon or youtube music (since both can be used on iOS and android). If I decided to commercialise the app then I would have to apply for a specific license from Spotify to do so. This is not the case and as such I do not have to worry about this application process.

With regard to the app being published on the app store, that is up to Apple, I will have to make my application completely compliant with their terms of service. Upon submission of an application, Apple normally take within 2 days to make their decision. I will be making these submissions of

the app to Apple before the preparation for my oral presentation, so that I can potentially demo the app. In the unlikely case that Apple are persistent in rejecting my app, then I must be able to simulate users using the device so that during the demo my complete app can be demonstrated.