A mobile application for democratic music playback

Author: Harry Verhoef

Supervisor: Till Bretschneider

Year of study: 3

Abstract

People in a social setting must typically rely on a single device to control music playback, that is to say it is unifocal, leading to potential conflicts of preference. This project entails the creation of a mobile application that allows users to democratically elect tracks to be pushed to a queue on a host device. Alongside this, a hybrid recommendation system consisting of genre and artist classification models is developed to recommend tracks to groups of users, which can then be voted for and consequently elected. The application encompasses a large variety of different technologies to provide a music playback environment that is arguably more democratic than any current solution.

Keywords— Software Development, Machine Learning, Democracy, Music

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 directly controlling what music is being played. As a result, many people are subject to music they are not particularly fond of. The aim of this project is to introduce the most democratic music playback environment possible, one where the majority of people listening are happy with what they are listening to.

First we will review any existing solutions to this problem, deducing how we can achieve a greater level of democracy. Then, in the methodology chapter, a project timetable will be laid out along with any project planning, and a product backlog will be established which will be motivated by the deductions made in the previous chapter. Following that, the process of designing and implementing the solution will be described in detail, in alignment with the project plan and with the objectives outlined in the product backlog. After that, the solution will be tested and evaluated. Finally, any future work will be listed and conclusions will be drawn.

Chapter 2

Review of Existing Solutions

There have been previous attempts at making music playback a democratic process. A good project will review these attempts and ultimately deduce what can be done to maximise the democracy of music playback, either by adoption or omission of certain features from these existing solutions.

2.1 Spotify Social Listening

In the June of 2019, Spotify introduced a social listening system [1] that allows users to join a "party". Once in the party, users have complete control over the party queue. This system is a feature of the Spotify application, and therefore requires no download of any additional package or application.

The mechanism by which users can join parties is through the activation of their on-device camera and scanning of a bespoke Spotify barcode as shown in figure 2.1. This is an efficient mechanism that allows users to easily join the party, very little effort is spared. Seeing as the system is a feature of the Spotify application itself, the user-interface will be very familiar to the user and the system, as a result, very easy to use. It's as if the user is using the regular Spotify application, except that other users have complete control over the queue too.

There are many limitations to this system. For one, all users part of the party must have a Spotify account, meaning that any users without one are left without any form of control over the music playback. Moreover, the system is more anarchic than it is democratic. There are no regulations or limitations as to what each user can do, they can all delete tracks off the queue, add as many as they want, change the order completely, etc.

Figure 2.1: Spotify Barcode

2.2 Festify

Festify [2] is a web application that allows users to connect to Spotify and create parties, then users can join the party by entering an alphanumeric party ID, all done while in a web browser. Once in the party, users have the ability to vote for the tracks they like and the most voted for will be played first. Festify relies on users to add songs to the queue (to be put up for voting), the creator of the party must specify fallback playlists which will be played in the event where no tracks have been queued.

The concept of allowing users to vote for tracks is inherently democratic; the track that gets the most votes and is therefore played next is only so as a direct consequence of combined user preference over the other tracks. Users can quickly join parties using the party ID, since it's short and memorable. By the same token, it is easy for users who are already in the party to reveal the party ID to users who are not.

However, users can vote for an unlimited number of tracks at a time, with different users voting more or less frequently, making it hard to ascertain the value of a single vote. This undoubtedly makes the process less democratic, since some users may vote very generously, and others more seldom, resulting in a substantial bias. Moreover, Festify provide the party creator with the ability to enter "Admin Mode", which gives the party creator permission to completely override any decisions made by the party. Conceivably, this is anti-democratic and self-defeating, it doesn't matter how seemingly popular a track may be among the party, if the party creator doesn't favour it they could skip or remove it from the queue. It can also be argued that the mechanism by which tracks are shortlisted is anti-democratic, since it is possible that a small minority of users are the only ones who shortlist the entirety of tracks put up for voting.

2.3 OutLoud

OutLoud [3] is a mobile application that allows users to create parties by connecting to Spotify and then uploading a single playlist. Users can join this party using a deep link. Similar to Festify, users can add and vote for tracks once they're in the party. Users can explicitly see what other users have voted for, and users who add tracks that get plenty of votes are shown on a leaderboard

as the "top DJs".

Seeing as OutLoud is a mobile application, available on the iOS App Store and the Google Play Store, it is available for a relatively large percentage of the population. Consequently, the application is more democratic, since more users have access to it and therefore user preference is better modelled. Users can down-vote tracks too, making the queue more malleable to user preference and thus more democratic. The "top DJs" feature encourages participation from users, meaning that they're more likely to vote.

The OutLoud application falls victim to many of the aforementioned pitfalls of Festify. Specifically, there are no limits on the number of tracks the users can vote for or against on, and the shortlisting mechanism can introduce bias. Furthermore, the up-votes aren't anonymous, rendering them as potentially less honest. The deep link that allows users to join the party must be sent to new users, and is not a particularly graceful procedure.

2.4 Deductions

A successful project will take the positives from the existing solutions and attempt to omit the negatives. In the context of this project, a positive feature is one that contributes to increasing the level of democracy in the decisions made regarding music playback, and a negative feature functions conversely. 8 key deductions can be made from the 3 highlighted existing solutions, as to specifically what will help this project be a success.

Instead of parties, the project application will use lobbies, since the proposed application is massively versatile in its usage. Users would be able to set up democratic music playback environments anywhere, not limited to any type of social setting. For example, road-trips, gyms, parties, radios can all make good use of a democratic listening environment.

1. A lobby-joining system that employs both IDs and barcodes, and does not require a Spotify account, so that users can quickly and easily join lobbies.

2. A similar user-interface to that which the user is already familiar with, to make the application intuitive to use.

3. An anonymous voting system where users can vote for only one track at a time, so that users give their honest opinion on their single favoured track.

4. Deployment of the application to a variety of platforms, making it more available and therefore more democratic.

5. Functionality that allows users to give negative feedback on tracks, making the party more malleable to user preference.

6. Functionality that ensures no user (even the party creator) has complete control over the queue.

7. An objective shortlisting mechanism that takes into account user preference, so that no bias is introduced.

8. Features will be provided to encourage participation from users.

Chapter 3

Methodology

3.1 Choice of Development Methodology

The principle of agile development will be used for the development of this project, namely since one of the fundamental aspects of agile development is a dynamic product backlog. This is important since the author has little experience in the development of such a project and as a result, requirements will likely need to be amended throughout the course of development as the author learns. If the development of the project were to be completed using a waterfall methodology, it is likely that the developer may outline impractical or infeasible requirements, which would be much more difficult to amend since the whole development procedure would have been planned rigorously.

Agile development itslef is a principle that defines a set of methodologies, and the methodology of choice in this case is scrum. Compared to the likes of extreme programming, scrum is a more general agile methodology and one that is perhaps more suited for a standalone developer.

3.2 Requirements Analysis

As part of the scrum development lifecycle, a product backlog must be established. The product backlog is an ordered list of everything that is known to be needed in the product [4], shown in table 3.1. These requirements were compiled as part of the project specification document given to the project supervisor on 09/10/2019, and were motivated as a consequence of the deductions made in section 2.4. The requirements are split in to 3 sections: Host, User and Miscellaneous. The "Imp." column represents the importance of the requirement, and is determined by how much the author perceives its satisfaction will contribute to the democratisation of the environment provided by the application. A chat room for each lobby will encourage user participation, and a recommendation system will be used for the objective shortlisting of tracks.

No.	Requirement	Imp.
1a	The host user will be able to create a lobby and invite users to said lobby by means of a key or QR code.	5
1b	The host user will be able to configure lobby settings before the lobby is created.	4
1c	The host user will be able to terminate their lobby at any time.	5
1d	The host user will be able to authenticate their account with the Spotify Accounts Services.	5
1e	The track with the highest vote score will be queued on the host device.	5
2a	Users will be able to join a lobby if they enter the correct key (or barcode).	5
2b	Users can only be in one lobby at a time.	3
2c	Users can leave a lobby at any time.	5
2d	Users can choose to show or hide a chat room if enabled.	2
2e	Users can choose to show or hide track lyrics if enabled.	2
2f	Users can thumbs-up or thumbs-down the current track.	4
2g	Each user has one vote for the next track.	5
2h	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 track they voted for.	4
2i	A user cannot obtain a vote weight of zero.	5
2j	If the volume control is enabled, users can vote to turn the volume up or down.	2
2k	Users on any iOS or Android Device can download the application from their respective app store.	4
2l	User votes will be anonymous.	4
3a	The backend will be scalable.	3
3b	The user interface will be responsive and familiar to users.	3
3c	The lyrics displayed on-screen will be synchronised with the audio being played at the time.	3
3d	The recommendation system will give lobbies recommendations on the next tracks to play and will be used as a tie-breaker when two or more songs have equal vote scores.	5
3e	The recommendation system will make use of objective user-preference analysis to infer suitable recommendations.	4
3f	The recommendation system will be hosted on an external application server.	3

Table 3.1: product backlog

3.3 Sprint Cycles

Perhaps the most principal component of scrum, sprint cycles will enable iterative development of the project application. The main components of each sprint will be sprint planning, design and implementation, followed by any testing, and finally demonstrating the progress made to the project supervisor. Select requirements from the product backlog will be placed on to the sprint backlog during the sprint planning stage. During design and implementation, the developer will attempt to satisfy the requirements on the sprint backlog. Suitable tests will then be carried in the testing phase, and any requirements on the sprint backlog that appear to have not been satisfied will be placed on to the product backlog, likely to be adopted by the following sprint.

3.4 Risk Management

"The objectives of project risk management are to increase the likelihood and impact of positive events, and decrease the likelihood and impact of negative events in the project." [5]. In the context of this project, positive events are the satisfaction of requirements on the product backlog. Conversely, negative events are those that hinder the satisfaction of such requirements. Thus, increasing the likelihood of positive events is achieved by decreasing the likelihood of negative events, or impact thereof. The potential negative events that could be encountered during the development of this project are summarised in this section alongside their management techniques, so that their impact is minimised.

3.4.1 Development is taking too long

Certain components may take longer than expected to develop and therefore the time constraint may become so large that it is likely a large proportion of high-importance requirements will not be satisfied. In such an event, it will be necessary to either remove or amend some low-importance requirements from the product backlog such that the time pressure is sufficiently alleviated to accommodate development of higher importance requirements.

3.4.2 Development equipment becomes unavailable

Development equipment may become unavailable for a wide range of reasons. For example, the development machine or the testing device may break. As a result, procedures that are dependent on the unavailable equipment may be delayed and the time constraint may increase. In such a scenario, the protocol detailed in section 3.4.1 would be employed where the lowest-importance requirements would be removed from the product backlog to accommodate the potential satisfaction of more important requirements. To further mitigate any negative impact such an event would have on the development process, Github, an external source-control service will be used to back-up the project code at

regular intervals. Therefore, upon replacement of the development machine, the code would be pulled from the project repository and development would continue.

3.4.3 Legal, social or ethical issues

If any legal, social or ethical issues were to arise it will become necessary to either amend or remove those requirements that these issues hinder such that there is no longer any issue. For example, in the unlikely event that Spotify were to revoke the authors ability to utilise the Spotify API (and they have every legal right to do so), then a large portion of requirements would be rendered unsatisfiable unless amended or removed. Instead, the project application would use the Apple Music API [6] and amend those requirements that depend on the Spotify API so that they now depend on the Apple Music API.

3.5 Project Timetable

The proposed timetable seen in table 3.2 shows the projected order of tasks that are to be completed, alongside their projected date. The purpose of this is to provide clear deadlines for each task so that the amount of time constraint at any given time is clear. Moreover, the proposed timetable will be compared to the actual timetable during project management evaluation, to provide an insight as to how well time was managed and how much the proposed and actual timetables differ. It is assumed that the oral presentation will take place on the earliest possible day, so that any unexpected increase in time constraint is avoided.

No.	Task	Date
1	Write specification	01-Aug-19
2	Project planning	03-Aug-19
3	Begin development	05-Aug-19
4	Submit specification	09-Oct-19
5	Submit progress report	25-Nov-19
6	Oral presentation	02-Mar-20
7	Finish development	09-Mar-20
8	Test application	23-Mar-20
9	Fix any failed tests	30-Mar-20
10	Submit final report	27-Apr-20
11	Complete further work	15-Jul-20

Table 3.2: Proposed timetable

3.6 Project Planning

3.6.1 Architectural Design

In order to begin the process of designing and implementing the fundamental components that compose an application, it is important to establish an architectural design so that the fundamental components and their interactions with each other can be deduced. This is a procedure that took place before any sprint cycles were initiated in a phase called "Project Planning". The product backlog dictates what features the architecture will accommodate, and thus helps to determine part of the architecture itself.

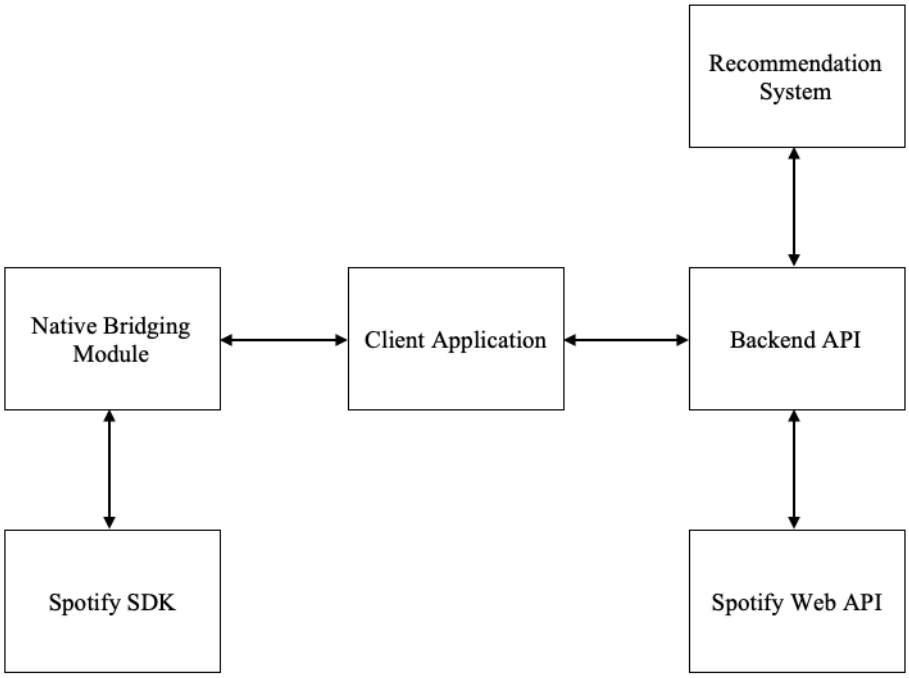


Figure 3.1: Planned Application Architecture

Requirement 1e specifies that the track with the greatest vote score will be added to the playback queue. In order to have access to the playback queue, the application must interact with the Spotify application (installed on the host user's device). This requires use of the native Spotify SDK, meaning an interface will need to be designed such that certain features of the SDK can be utilised to the application's benefit. This interface will be referred to as the native bridging module throughout this report, since it will have to be written in native code (because the SDKs are) and it will act as a bridge between Spotify application functionality and the project application. The backend API will interact directly with the client application, sending and receiving lobby

and user-specific data. The backend API will need to interact with the Spotify Web API for authentication purposes, as is necessary to satisfy requirement 1d. As is necessary to satisfy requirement 3g, the backend API will also need to communicate to and from the recommendation system, generating the shortlist for each lobby and sending it back to the client.

3.6.2 Technologies Used

With the architectural design established, it is possible to deduce the types of technologies that will be required to fulfil such an architecture. Furthermore, it is useful to define the specific technologies that will be used before any design and implementation has begun. It is much more challenging to integrate these technologies if the developer is unaware of any technology-specific limitations that may hinder said integration process. Owing to the architecture, there are 4 areas where different technologies will need to be applied.

React Native

React Native is a mobile application framework built by Facebook that is based on React.js yet compiles down to native code for iOS and Android, meaning it fully satisfies requirement 2m. React Native was chosen among other choices since it is a technology unfamiliar to the author and thus offers an intriguing opportunity to learn a new skill. Perhaps another massively important feature of react native is the ability to incorporate native modules through their built-in native modules library, which will massively ease the integration process between the native bridging module and the client application.

Objective-C and Java

The native code that is to be used for the development of the native bridging module is Objective-C and Java, for iOS and Android respectively. The reason behind these decisions is that the corresponding Spotify SDKs are written in these languages and therefore using anything else would add unnecessary complexity during integration. If this were not the case, then Swift would be used instead of Objective-C since it is a much more modern language and is syntactically familiar to the author, unlike Objective-C.

AWS Lambda Node.js Functions

Another JavaScript framework, Node.js was chosen for the backend API. Amazon Web Services (AWS) provide an event-driven hosting service called lambda, which translates small snippets of source code into executable lambda functions. Once lambda is integrated with another AWS feature, API Gateway, a lambda function can be called following API endpoint invocation. This architecture massively modularises the backend by separating each of its composing features. By doing so, the development and unit testing of each feature is made

much easier, which is important since node is another framework that the developer has no experience in. In 2019, apple introduced App Transport Security (ATS), which meant that no application downloadable from the iOS app store could communicate with an external server using simple HTTP. Each request to an external server has to be made using HTTPS, which requires an SSL certificate. By using AWS instead of an alternative cloud service, the developer will easily be able to set up an SSL certificate free of charge, a feature that most other services charge for.

Tensorflow

Tensorflow, built by Google, has been the staple deep-learning framework for a long time now. There is extensive documentation and support for tensorflow. Crucially, AWS SageMaker is a service that supports the hosting of pre-trained tensorflow models (which will be trained on a local machine to avoid costs). Thus, the interactions between the recommendation system and the backend API will integrate seamlessly, as they will be contained within a single AWS environment.

Chapter 4

Design

Each section in this chapter represents a fundamental component of the application, and describes the process of designing it.

4.1 Preliminary User Interface

The user interface is the last fundamental component that is to be fully developed, seeing as it does not in essence add any critical functionality to the application. However, a preliminary UI is to be developed with the sole function of supporting the development of more crucial functionality. At the final stage of the development of the application, a new and more complex final design will be adopted by the UI, which intends to adopt conventional design patterns to help satisfy requirement 3b.

4.1.1 Component Tree

Before beginning the design process preliminary UI, it is first important to establish a constant foundation upon which it will be developed. In figure 4.1, each node in the component tree represents a react.js component, exported by a JavaScript file. The edges shown between components do not constitute relationships of inheritance, merely adjacency. Each of these components mimic the pages that users may "visit". Users will be able to visit these pages by means of a stack navigator, similar to that which a web-browser employs. Users can visit any component adjacent to the one they're currently in. By traversing deeper into the tree, the component is pushed on to the navigation stack. If the component at the top of the stack is popped, then the user will go "back" and traverse upwards through the component tree.

Landing.js

The purpose of the landing component is similar to that of the index.html on a website. The Landing component is the component the user will "land" on

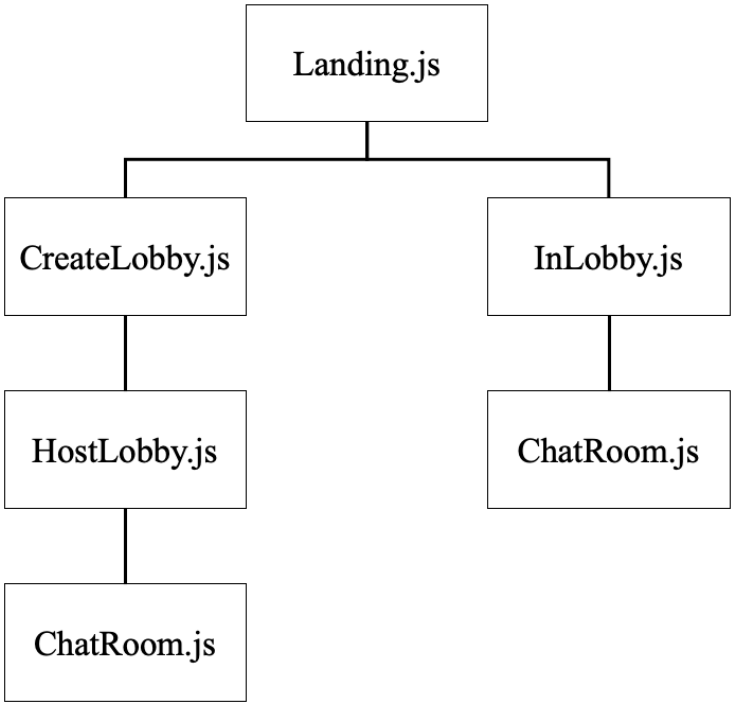


Figure 4.1: React.js UI component tree

when they initially load the application. It is the default component. From the landing component, the user can either join a lobby or create one. This is further demonstrated by figure 4.1, where the only 2 components adjacent to the landing component are the CreateLobby and InLobby components. Users cannot go "back" from the landing component since there is no component to go back to, there must always be at least 1 component on the navigation stack.

CreateLobby.js

The CreateLobby component provides functionality enabling a future host user to tweak the settings of their future lobby, before any user can join. Specifically, the future host user will be able to alter all the lobby settings described by requirement 1b. It is also the responsibility of only this component to authorise a Spotify account, since only users who wish to create a lobby need a Spotify account (with the Spotify app installed).

InLobby.js

Once a user has successfully joined a lobby, the InLobby component will be pushed on to the navigation stack and the user will be able to access lobby

functionality instantly. From this state the user can either visit the chat room or they can go back to the Landing component.

HostLobby.js

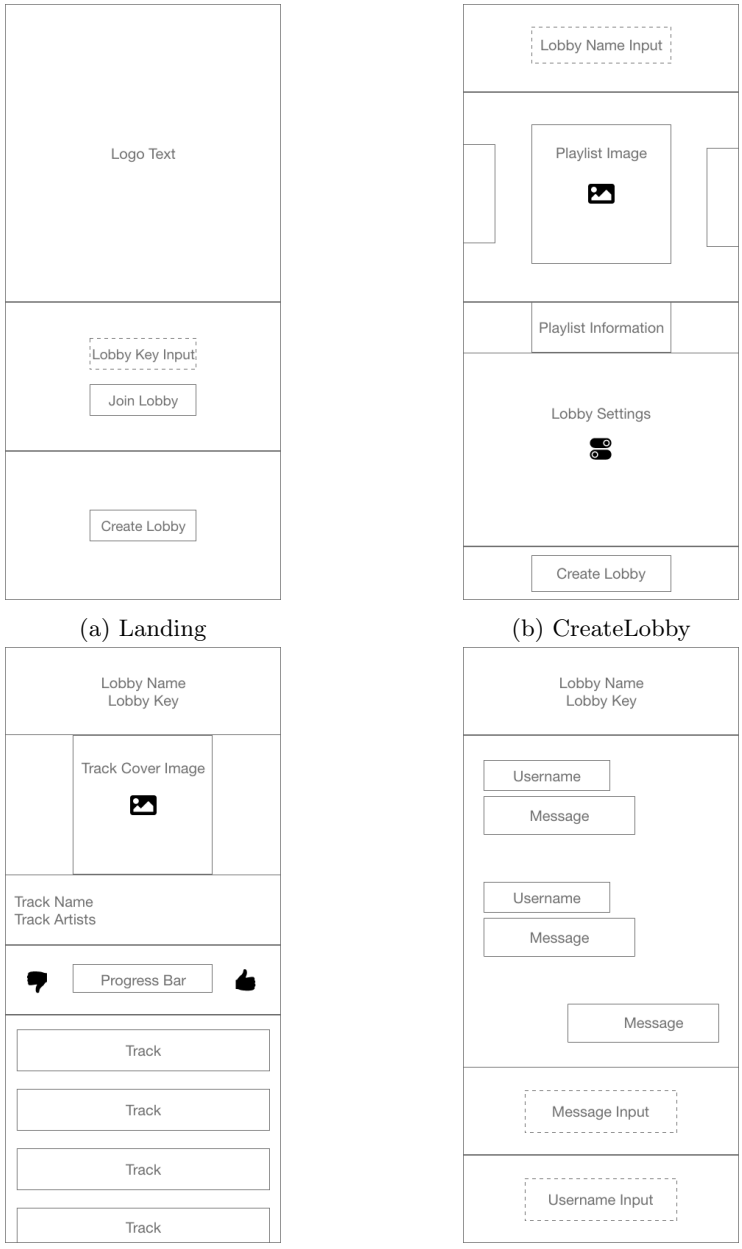
Host users will have the HostLobby component pushed on to their navigation stack once they've finalised their lobby settings and confirmed that they're ready to instantiate it. At this point, users will be able to join the lobby. Host users will be able to visit the chat room just like users with CreateLobby at the top of their stack. However, if host users have the HostLobby component popped off their stack it will leave CreateLobby at the top. Meaning, if the host user is to go back, they will be shown the CreateLobby interface, where they can create a new lobby or go back to Landing.

ChatRoom.js

The function of the ChatRoom component is to provide users with the ability to communicate with other users in the lobby. Users will be able to set their username, though it is not necessary as they will be given a unique guest username by default. Either way, users will be able to type their messages and send them to the entire lobby. It is worth noting that the chat room is a feature that may have been disabled prior to the instantiation of a lobby, by the host user. In which case, users will simply not be able to enter the chat room.

4.1.2 Wireframes

As previously mentioned, the sole purpose of the preliminary UI is to accommodate crucial functionality. As such, there will be little thought put into good design practice. After the crucial functionality has been developed, a review of the preliminary UI can be done to develop a new, final UI that is more aligned to the requirements analysis and general design principles. Figure 4.2 shows the wireframes for the components of preliminary UI, designed in Adobe XD. Plenty of detail is left abstract, such as colour scheme, since the style of the application is not important at this stage. Note that the wireframe shown by figure 4.2c is representative of both the HostLobby and InLobby components. This is done to highlight the fact that the host user will have no functionality that is not made available to other users in the lobby. They're different components only due to differences in the life-cycle; the HostLobby.js component will be interacting with the backend much more than the InLobby.js component. The differences in interaction will not be made explicit nor available to users, since the UIs are identical and no distinguishable functionality can be invoked by the user, only the device. In order to navigate backwards (to pop the current top component off the stack), the user must swipe from left to right. The wireframes are designed on the canvas of an iPhone X. While the device that the application will be developed on is an iPhone 8, this should not matter since the preliminary UI is simple and a responsibility of even the preliminary UI is to be responsive.



(a) Landing

(b) CreateLobby

(c) HostLobby and InLobby

(d) ChatRoom

Figure 4.2: Preliminary UI Wireframes

4.2 Native Bridging Module

The Native Bridging Module acts as an interface between Spotify functionality and the application itself. As such, in order to design it the functionality the application requires will need to be specified.

1. An authorisation method will be required that will perform an "app-switch" to the Spotify Application in order for the user to log in and verify that they accept the project application will be able to use their account in certain ways.

2. A play method will be required to invoke instant music playback of a specified track. This method will take the Uniform Resource Identifier (URI) in order to identify the track to be played.

3. A queue method will be required to add a track on to the playback queue, when it is either voted for, or inferred by the tie-breaking feature of the recommendation system.

4. Finally, a getAccessToken accessor method will be required to obtain the user's access token, which in turn can be used to send a multitude of requests to the Spotify Web API. Therefore, this method will likely be invoked before any interaction with the backend API that requires Spotify data, such as querying track information.

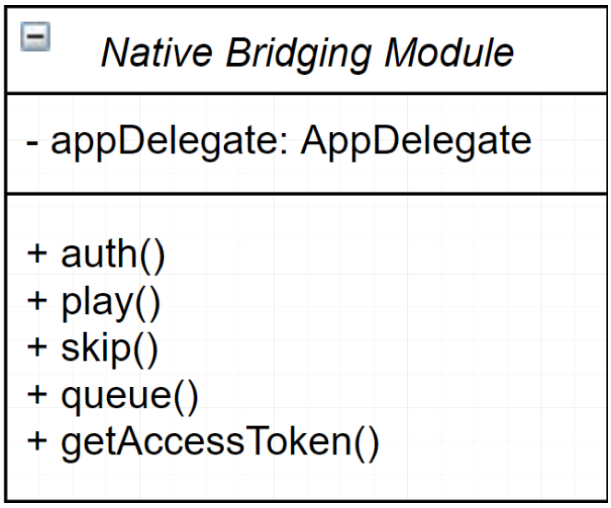


Figure 4.3: Class Diagram of the Native Bridging Module

Besides these public module methods, which are accessible to the client application, there will have to be a variety of private native methods. The purpose of which is to collectively handle both the beginning of the authorisation cycle,

and the consequent Spotify session that is created upon successful authorisation, as detailed in the Spotify SDK documentation [7].

4.3 REST API

There is plenty of need for stateless communication between the client and server in the project application, for example a new user joining a lobby will need to send an initial HTTP request to the API in order to obtain the currently playing track data. Listed below are the endpoints (resources) which compose the REST API, and their corresponding methods (GET/POST).

POST /delete_lobby

This endpoint will be called upon by the host device when the host user decides to terminate the lobby they created. The endpoint will first kick every user (including the host) out of the lobby then delete all of the lobby data such as votes or recommendations. The functionality that this endpoint will provide is motivated by requirement 1c.

GET /get_lyrics

The sole purpose of this endpoint is to utilise a currently unknown API to gather the lyrics data for a given track, which can then be synchronised with playback on the host device (requirement 3c). This endpoint will be invoked by each device in the lobby, every time a new track is being played by the host device. As such, there will be no need to store the lyrics data for the currently playing track in the database.

GET /get_next_track

This endpoint is used to calculate the next track, determined by user votes and if need be, the tie-breaking recommendation system. The response generated by this endpoint will be used to queue the corresponding track on the device, using the native bridging module. It is invoked only by the host device, since it is the only the device in the lobby able to interact with the Spotify playback queue. The functionality this endpoint delivers is necessary in order to satisfy requirement 1e.

GET /get_playlists

Much thought was put into the logistics behind obtaining user playlist data. The data is necessary to allow the host user to select the lobby's base playlist (requirement 1b). The are two ways to obtain this:

1. Through the native bridging module, it would be possible to create a new public method that will utilise the Spotify SDK to acquire user library data in the form of an n-ary tree. Upon traversal of this tree, it would perhaps

return a subtree whereby each leaf represents a native playlist node object. Such a solution would be generally much faster than any API request (due to network latency), and would work offline. However, in order to retrieve and then use objects from the native bridging module, these objects must first be converted into non-native representations. There is support for the most simple objects; an `NSMutableDictionary` will be converted into a JavaScript object. However, there is no support for the conversion of a `UIImage` to an `RCTImage`, meaning the cover image of the playlist cannot be exported to the client application from the native bridging module.

2. The second solution is to use the Spotify Web API to obtain a temporary URL for the cover image, which can then be stored and used to render the image for every user in the lobby. This solution is perhaps more timely yet gives users the ability to see the cover image, making the UI more recognisable (requirement 3b). Moreover, using this solution, no actual image needs to be stored, only the URL, making this solution much more space efficient.

Through this evaluation, it is clear that the creation of an API endpoint is the better design pattern, since it contributes to the satisfaction of two requirements instead of one.

GET /get_recommendations

This endpoint serves to invoke the recommendation system, gather the inferences made and use them to generate a set of recommendations which will then be stored in the database. At which point, the API will send a status 200 (OK) response back to the client, with no recommendations as part of the response. It is an important design decision to separate the invocation of the recommendations system with the retrieval of its output. The reason for this is that the invocation only has to be done once per track being played for each lobby, yet the recommendations are constant lobby-wide and thus need to be accessible instantly and without invocation, to all users. Therefore, it makes sense to have one device invoke the recommendations system at some point just before a new track is elected, and after that every device can request the result of said invocation. The Spotify Web API will be used in collaboration with the recommendation system to generate the recommendations, in which case an access token will be needed alongside the HTTP request. The host device is only device in the lobby guaranteed to have a stored access token which can be used in the Spotify Web API. As a result, this endpoint is only to be called by the host device.

GET /get_stored_recommendations

Since the recommendations, once generated, are stored on the database, each device will at some point need to retrieve these recommendations. This endpoint allows all devices to retrieve the result of the invocation recently executed by

the host device. Therefore, according to the application lifecycle (see figure 4.7), each device will invoke this endpoint once the next track has begun playing.

POST /join_lobby

This endpoint will provide users with the ability to join a lobby, and in doing so it will respond with data regarding the current lobby state (current track, current votes, etc.), fetched from the database by a join_lobby lambda function. This endpoint will satisfy requirement 2a fully.

POST /make_lobby

Invoked by only the host device, this endpoint creates a lobby and is only to be used following successful Spotify authorisation. Once this endpoint has been successfully invoked, any user will be able to join the lobby using the aforementioned /join_lobby endpoint. This endpoint is necessary as per requirement 1a.

POST /refresh

OAuth2 is the authorisation procedure used by Spotify Accounts Service, and this endpoint is mandatory in order to have the application refresh access tokens (using a refresh token) once they have expired. Moreover, it is crucial that this endpoint is implemented early on, since it quickly generates new access tokens, which are essential for any unit tests involving Spotify Web API. Outside of testing purposes, this endpoint will be invoked by the host device from within the native bridging module (specifically the getAccessToken method) if the access token for the session has expired. This endpoint is motivated by requirement 1d.

POST /set_track

This endpoint is to be invoked by the host device, once either the play or queue native bridging module methods have been called. The product of this endpoints invocation is the execution of a lambda function which stores the track data as in the database. Following invocation, if a new user is to join the lobby (that is to call the /join_lobby endpoint), the track that is displayed upon joining will be up to date.

POST /swap

Another authorisation endpoint, the /swap endpoint is necessary to swap multiple parameters such as client_id and client_secret for a refresh and access token. This endpoint is invoked every time a new session is created by the native bridging module. Essentially, it is invoked every time a new user wishes to authorise their Spotify account. Again, this endpoint jointly contributes (along with /refresh) to the satisfaction of requirement 1d.

POST /thumbs

The thumbs endpoint is called once any user has changed the state of their feedback for the currently playing track. It can be neutral, up, or down. The state is reset to neutral after every track, as part of the /set_track endpoint. This endpoint is motivated by and fully satisfies requirement 2f.

4.4 WebSocket API

Certain features of the application such as persisting lobby state and voting are very stateful and therefore a REST solution is infeasible. It would be possible to have a timer on each device sending an HTTPS request recurrently until the lobby is terminated. However, this is only mimicking stateful communication and there are protocols that actually accommodate it. WebSocket is a protocol that provides stateful full-duplex communication over a TCP connection. Importantly, WebSocket interactions between client and server allow the server to send data to the client without first receiving a request. React Native comes with built-in WebSocket support, and AWS recently introduced WebSocket APIs as a separate API that can be developed as part of API Gateway. Therefore, the WebSocket and REST APIs will be completely separate. Instead of endpoints, the WebSocket API utilises "routes", but at a high-level they function similarly. The WebSocket API will be composed of 5 routes, those that are prefixed by a "$" are routes that API Gateway requires. Each route has a corresponding lambda function that is called whenever a message is sent to the route.

$connect

On the opening of a WebSocket TCP connection between the client and the server, this is the route that is messaged and will serve to store said connection. Once the connection has been established, the server can communicate with the client without the need for the client to send a message in the first place, until the connection is closed. This route is messaged by any user who has just joined or created a lobby.

$disconnect

Conversely, the purpose of this route is to terminate the connection instantiated by the $connect route. This route will be messaged once a user has left a lobby or left the application.

$default

In the case where a WebSocket connection attempts to transmit a message with a route that is not defined by the API, then it will fall under this route. In such a scenario, the API will simply respond with a message stating that the route is invalid.

vote

When any member of a lobby votes on a track, this needs to be updated for every other member of the lobby too, since it is crucial that the entirety of the lobby can see the state of the votes in real-time. Therefore, this route will take a user vote, add it to the current state of votes, remove any previous vote that the user has made, and then push it as a message to all users part of the same lobby.

next

As previously mentioned, each lobby has state, and part of that state is the currently playing track. Each user will need to have their lobby's state updated once a new track is being played on the host device. Once the host device detects that a new track is being played, it will send a message to the WebSocket API along the "next" route stating the currently playing track. Then, the API will send a message to each of the users of the lobby with the same information to suggest that the new track is being played.

4.5 Database

Given that the backend is hosted on an AWS environment, in order to reduce backend internal latency, the decision was made to use an AWS-hosted database too. There are plenty of different Database Systems that the backend could utilise, but due to the application's scalability requirement (3a), a schema-less database called "DynamoDB" was decided upon. Since DynamoDB is schema-less, it provides the application with much more flexibility in its data storage. Particularly, unique device ids are represented in different ways on different platforms, meaning that only one device id field is required in a schema-less database. Yet, more may be required in one that has a strict schema, to accommodate the variety of id formats. The database will be composed of 4 distinct tables, as follows:

1. **Device** - This table will store information regarding each device, such as the device id, the device's vote-weighting, the lobby that the device is in, etc.

2. **Lobby** - This table will store lobby data, such as the lobby key, the lobby length, the lobby name, etc.

3. **Lobby-Connection** - The purpose of this table is to be able to group the connections of the lobby together so that WebSocket messages can be sent to the lobby as a whole. The lobby attributes are lobby keys and the connection attributes are connection ids.

4. **Lobby-Track** - Similar to the Lobby-Connection table, this table makes use of a composite primary key so that for each lobby, a votes object can

be constructed in O(1). The lobby attributes are lobby keys and the track attributes are track ids. There is also a user_list attribute that contains a list of device ids that have voted for that specific track in that specific lobby.

4.6 Recommendation System

Since the developer has no experience of developing any form of machine-learning system, let alone a recommendation system, the initial design would have to be based off research. In order to know what to look for in terms of the research, it is important to understand the inputs that can be taken from the system design thus far. The key inputs, per lobby, that the recommendation system may be able to take are:

1. **User Preference** - Using the thumbs-up/thumbs-down feedback system, it is possible to determine user track preferences. Moreover, it is possible to derive user genre preferences and user artist preferences, since both of these are components of a track.

2. **Lobby Name** - The name that the host user has given to the lobby may be able to provide inferences as to the type of lobby. For example, if a Warwick Gym employee is setting up a lobby for the gym, they may use "Warwick Gym" as their lobby name, leading to potential inferences about the type of music that is relevant for that social setting.

There could be an argument made that another important input is the base playlist. The base playlist will have a name, which could again be useful for inferences, there is also likely to be a variety of tracks as part of the playlist, which can be used to establish an approximation as to what music the lobby will elect. However, the purpose of this project is to make the music playback process as democratic as possible. Making use of a saved playlist in the host user's library introduces substantial bias, when generating recommendations. While the lobby name is also written by the host user, it is intended for the entirety of the lobby. The same cannot be said about a playlist on the host user's Spotify account. User preference is a mandatory input of the recommendation system as per requirement 3e.

The recommendation system that is required for this project falls under a similar category to Automatic Playlist Continuation (APC) systems, which suggest appropriate tracks to add to playlists. The ACM RecSys Challenge 2018 [8] led by Spotify gave teams a dataset of 1 million playlists, along with metadata and asked them to build an APC system. In 2019, a paper analysing the solutions by each team was published [9]. Interestingly, in this paper it is described that most successful solutions consisted of a combination of different models. Lots of successful teams used a model for cold-start situations, attempting to infer tracks to recommend based on only the playlist title. Specifically, the paper states that convolutional neural networks can be useful for "extracting useful

information from playlist titles". Also, the paper suggests that a number of successful teams utilised Recurrent Neural Networks (RNNs) for "modelling the sequence of tracks in a playlist". From this it is possible to link these two networks to the 2 previously established inputs. A convolutional neural network could be used to extract information from the lobby name, and a recurrent neural network could be used to model a sequence of tracks ordered by user preference, since music playback is also a sequential process [10].

Both of these models are most likely to be given unseen data at the point of prediction, since the data that they will be trained on will undoubtedly be a tiny subset of the possible inputs. Consequently, the objective when training these two models is to maximise the validation accuracy.

4.6.1 Genre Classification Model

Using the lobby name, it is infeasible to suggest that specific tracks can be inferred; there are over 50 million tracks [11], yet there are only 126 genres that can be used as recommendation seeds [12]. Clearly, a genre classification model is more realistic.

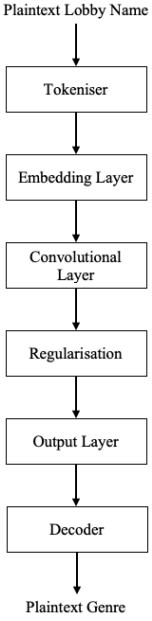


Figure 4.4: Genre Inference Process

The basic architectural design plan of the network is to include a 1-dimensional convolutional layer which will extract the defining features of the lobby name, in this case it will extract the words that the resultant inference depends on most. Before the convolutional layer, an embedding layer will be required to produce an embedded representation of the input (the tokenised lobby name). A dropout will be applied in an attempt to mitigate any overfitting that may occur, and thereby increase the validation accuracy of the model [13]. Since this is a Natural Language Processing (NLP) model, prediction inputs will need to be tokenised, and the output may need to be one-hot-decoded, since this is a multiclass classification network. Any other specific details of the model are superfluous to outline pre-implementation, since it is incredibly likely that said details will change once the model is tuned.

4.6.2 Artist Classification Model

Unlike genres, the number of artists available through Spotify is massive, yet artist inference is still preferred over specific track inference. This is because the output of the recommendation system will be used as a seed for the Spotify /recommendations endpoint and a track is too specific. Moreover, an artist inference compliments a genre inference better than a track inference, since a track belongs to one genre and an artist can have many tracks belonging to many different genres. However, while distinct identifiers for each track may be too specific, track-features such as acousticness, valence, and danceability produce a form of latent representation of a track. This latent representation can be used alongside an artist identifier to describe a track, with the benefit of less data being required. Additionally, when using track-features to describe a sequence of tracks, general user mood can be modelled and specific track features may render themselves helpful in the inference process. For example, if a user is listening to tracks with high tempo, danceability and liveness, it would make sense to infer artists that generally contain tracks with similar features. The model would be able to learn to attribute certain artists with the latent representations of their tracks, making the concept plausible. The Spotify Web API contains an endpoint /audio-features that takes a track id and returns a set of audio features, including those mentioned already.

User preference is a vague concept and there is no definitive way to calculate it. In the context of this specific application, user preference represents the collective feedback of all users in a lobby. One feature of RNNs that use back-propagation is that elements at the beginning of the input stream have less effect on the output. This is known as the vanishing gradient problem and its impact is minimised through the use of a Long Short-Term Memory (LSTM) network, a solution that many teams successfully employed in the 2018 RecSys challenge [8]. While most see this as a problematic inescapable feature of RNNs, it can be exploited to help process user preference in a manner such that the best received tracks are placed at the end of the input stream and the worst at the front. Input streams can be represented as vectors or matrices, where the elements at the front of the stream are at the top.

$$\vec{w} = \begin{bmatrix} 0.80 \\ 1.50 \\ 1.00 \end{bmatrix}, \vec{r} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \tag{4.1}$$

$$w_i = w_i + \max(-\frac{1}{2}w_i, \frac{1}{n}\sum_{j=0}^{n} r_j) \tag{4.2}$$

$$r_i = \begin{cases} 1 & \text{user } i \text{ gave thumbs-up} \\ -1 & \text{user } i \text{ gave thumbs-down} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

Vectors $\vec{w}$ and $\vec{r}$ as shown in 4.1 represent examples of vote-weight and rating vectors in a given lobby. For each user i in said lobby, w_i denotes the user's

vote-weight. That is the amount that their vote is worth, but can also be used as a measure of user credibility, determined by the rest of the lobby. For the same user i, r_i denotes the rating that user i has given to the currently playing track, determined by the thumbs-up/thumbs-down feedback system. The vote-weight adjustment demonstrated by equation 4.2 is applied to any user i who voted for the track that is currently playing, and is applied at the end of the track. In 4.2, n represents the total number of users in the lobby. Equation 4.3 Demonstrates the derivation of r_i, it is worth remembering that the thumbs-up/thumbs-down status is reset once each new track begins, meaning r_i is also reset.

$$\vec{r}^{T} \cdot \vec{w} = x_t \tag{4.4}$$

$$\vec{a} = \begin{bmatrix} \text{Xyo4u8uXC1ZmMpatF05PJ} \\ \text{5K4W6rqBFWDnAN6FQUkS6x} \\ \text{15kaWg9fBJWfcbpSlHhOML} \end{bmatrix} \tag{4.5}$$

$$F = \begin{bmatrix} 0.31 & ... & 0.43 \\ 0.39 & ... & 0.47 \\ 0.41 & ... & 0.51 \end{bmatrix} \tag{4.6}$$

$$(\vec{a}, F) \underset{\text{prediction}}{\rightarrow} \hat{a} \tag{4.7}$$

Once a track t has concluded, the adjustment 4.2 is applied to the users that voted for track t, and then the track weighted-ranking x_t is calculated using equation 4.4. The weighted-ranking is then compared with the history of track weighted-rankings for that lobby. As demonstrated by 4.5 and 4.6, the artist id and track features of the 3 tracks that have had the highest weighted-ranking in the history of the lobby are used to construct a vector $\vec{a}$ and matrix F. Each $\vec{f_t}$ is a row vector of 8 track features: acousticness, danceability, energy, instrumentalness, liveness, loudness, speechiness and valence. Each a_t is a Spotify artist id. $\vec{a}$ and F are ordered in ascending order of track weighted-ranking x_t to exploit the aforementioned vanishing gradient problem. A heterogeneous pair $(\vec{a}, F)$ will be constructed, which will be taken by the RNN to generate a prediction in the form of an encoded artist id $\hat{a}$, as demonstrated by 4.7. This prediction process can be described in further detail by figure 4.5, where the design of the model layer architecture is also depicted.

4.6.3 Obtaining rankings

Instead of each model predicting one class at a time, ideally, the models could return a set of probable predictions, which would be more useful to the application, since there is a greater chance one of the predictions is true. In consequence, there is a greater chance that the recommendations are those that the lobby will vote for. Since the class labels for each model are one-hot-encoded, the softmax activation function will be used on the output layer. This activation function will return a list of probabilities ranging from 0 to 1. Instead of taking the index

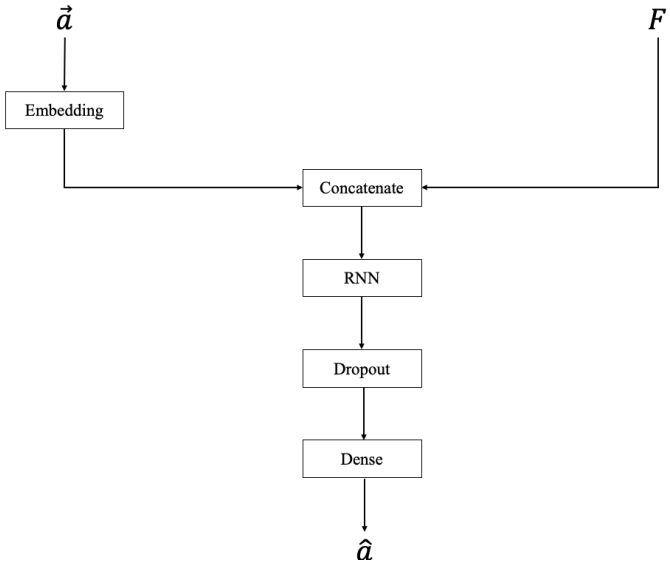


Figure 4.5: Artist Inferenece Architecture

that corresponds to the class with the highest probability, the application can take the top n indices.

4.6.4 Combining Models

The lobby name is not going to change after the lobby is created, and therefore neither will the inference made by the genre model since it is deterministic. As a result, the inferred genre seeds will only be used in cold-start scenarios, specifically where less than 3 tracks have already been played. The predicted artist seeds can be utilised any time after 1 track has been played, so for the 2nd and 3rd track of the lobby, both of the inferences will be used to generate recommendations. The decision making process can be seen in figure 4.6 in the form of a flow chart, where L represents the number of tracks that have been played in the lobby. Since the /recommendations endpoint can only take a maximum of 5 genre and artist seeds in total, each model will return the top 3 most probable seeds. Meaning that after the first and second track of the lobby conclude, the /recommendations endpoint will be given 5 seeds, then 4 after the third track, and 3 thereafter, due to the previously mentioned mechanism by which the models are combined.

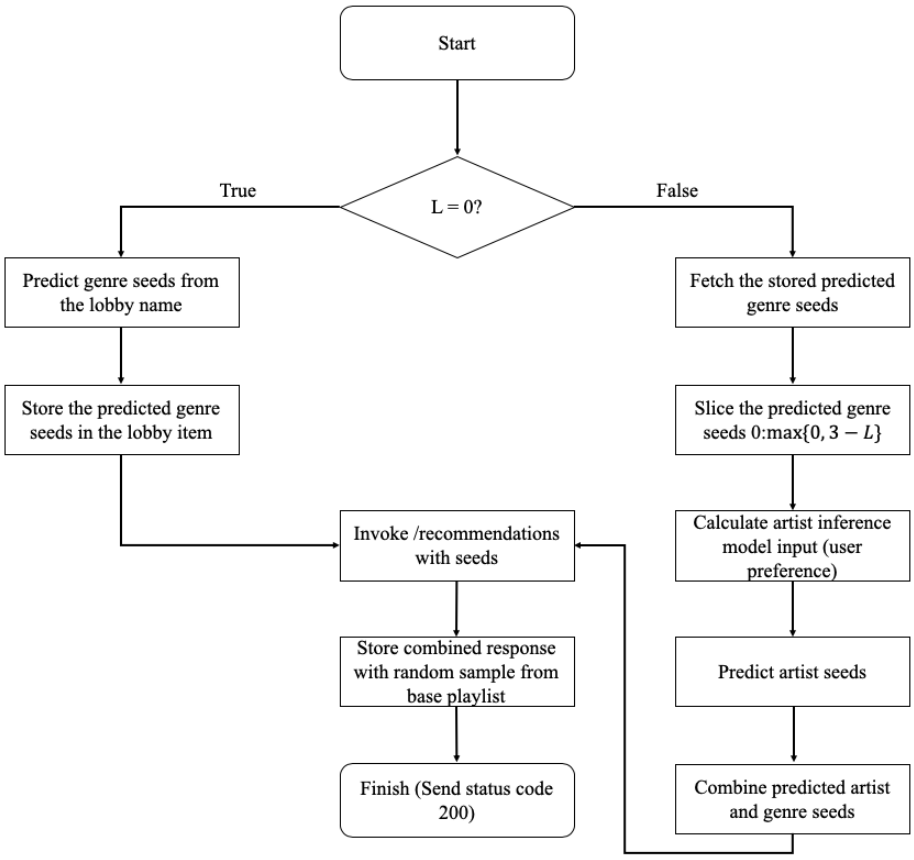


Figure 4.6: Recommendation System Flowchart

4.7 User Stories

In the interest of clarity, the sequence diagrams shown as figures 4.7 and 4.8 have abstracted certain procedures that play a part in the processes demonstrated. The Device lanes encompass that of the user, client application, and native bridging module. Any trivial database or Spotify Web API interactions such as the persistence of voting data or querying user playlists have been omitted from the "Backend API" lane. Moreover, neither sequence diagram denotes the entire domain of functionality available to the subject device in the context of the application, only a subset. The sequence diagrams represent example use-cases; The process of generating artist inferences is omitted since it is very similar to the genre inference process in figure 4.8.

Figure 4.7 demonstrates a sequence diagram for an example non-host user who joins a lobby, gives feedback on the currently playing track in the lobby (thumbs up/down), then votes on the next track to be played. 3 other users

in the lobby vote while this user is in the lobby, and at the point where the track concludes, the device receives a "next" message from the WebSocket API (invoked by the host device). Once received, the currently playing track information will update, and the votes, thumbs status, and progress bar will reset to their initial and default state.

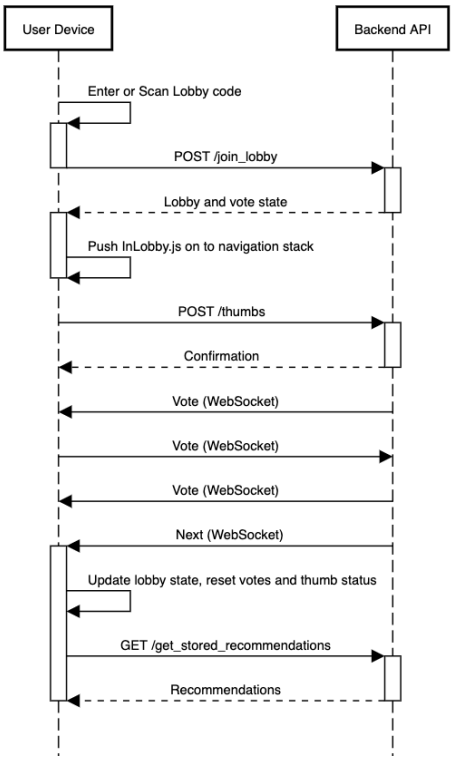


Figure 4.7: Sequence Diagram for an example non-host user

Figure 4.8 is more complex as it shows an example use-case for a host-user, who creates a lobby. The beginning of the diagram shows the user proceeding through the authorisation process; Authorising the application to use their account, then connecting the application to the Spotify app. Once this is completed, the user can choose the base playlist for their lobby, through the invocation of the /get_playlists endpoint which queries the Spotify Web API (not shown). Once the user confirms the lobby settings, they create the lobby. At this point, any user can join the lobby and begin voting for tracks. Also demonstrated is a vote coming in from another user, which becomes the most voted-for track and is therefore elected. After the currently playing track is finished, the elected track will play and the host-device will alert the backend API via both the /set_track endpoint and a WebSocket "next" message. The /set_track endpoint is for those users who are yet to join the lobby, and the "next" message is to alert those who have already joined.

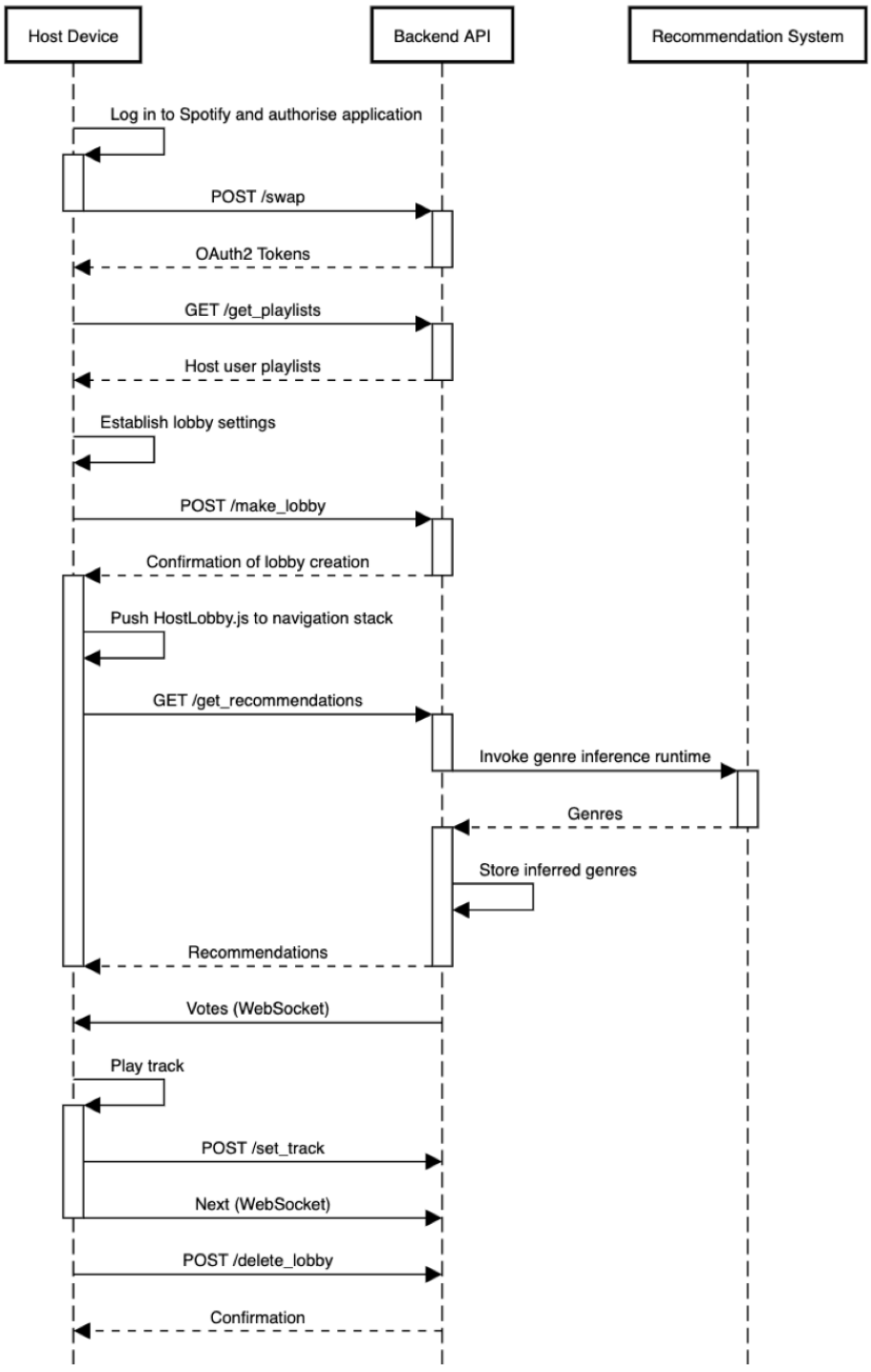


Figure 4.8: Sequence Diagram for an example host user

Chapter 5

Implementation

5.1 Overview

This chapter contains sections which describe the implementation process of each fundamental component within the application, including any unexpected problems that arose and the tackling thereof. There were 10 sprint cycles of varying length, starting from 05-Aug-19 and concluding on 09-Mar-20, as shown in table 5.1.

5.1.1 Amended requirements

Over the course of development, various requirements were either removed or amended from the product backlog and some appended to the list of future work as a product of the activation of any of the contingency plans highlighted in section 3.4 as part of risk management.

After it was discovered that lyrics could not be displayed without legal issues arising, requirement 2e was amended to avoid such legal issues by displaying track information instead of lyrics. Consequently, requirement 3c (synchronisation of lyrics) was completely removed from the product backlog since it was entirely dependent on the satisfaction of requirement 2f. Requirement 2f was later appended on to the list of future work since its initial implementation was deemed unreliable and it is a relatively unimportant requirement.

At the point of the progress report, requirements 2d and 2j were added to the list of future work, these represent the chat room and volume control requirements respectively. Also at the point of the progress report, requirement 2k was amended such that the application was to be built to only iOS devices. These decisions are a result of the activation of the contingency plan to ease time pressure, described in section 3.4.1. Later on, the amended requirement 2k was appended to the list of future work in another bid to alleviate some of the increased time pressure caused by delayed development.

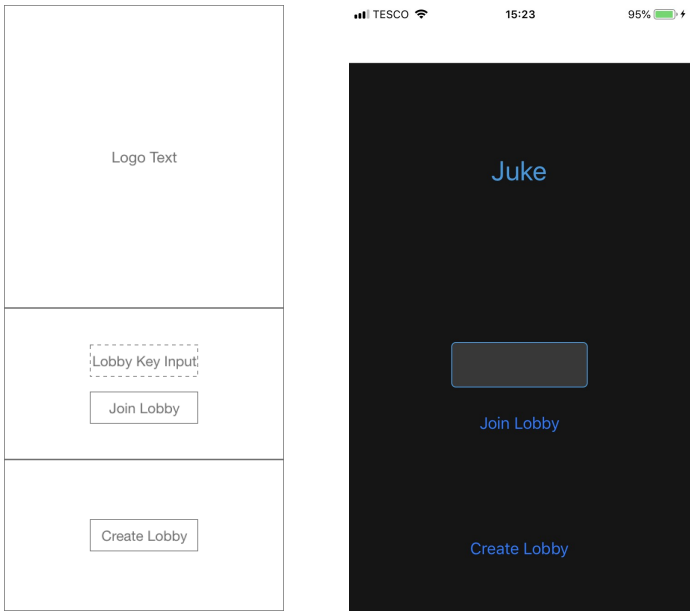
No.	Description	Weeks	Date Finished	Dependencies
1	Setting up the development environment.	1	12-Aug-19	-
2	Developing the preliminary UI and researching react-native.	2	26-Aug-19	1
3	Developing and integrating the native bridging module.	8	21-Oct-19	1
4	Developing the basic REST endpoints locally.	5	25-Nov-19	1,3
5	Researching AWS and developing the DynamoDB database.	4	23-Dec-19	4
6	Deploying the WebSocket API and most of the REST API on AWS.	4	26-Jan-20	4,5
7	Researching machine learning and training models	4	23-Feb-20	-
8	Deploying machine learning models and integrating with REST API	0.5	26-Feb-20	6,7
9	Final UI partially implemented for oral presentation	0.5	02-Mar-20	1,2
10	Final UI fully implemented	1	09-Mar-20	1,2

Table 5.1: Implementation timeline

5.2 Preliminary User Interface

The preliminary UI proved to be sufficient in order to aid the development of the rest of the application. The wireframe designs and the implementations only differ in resolution since the wireframes use the iPhone X as a canvas and the device used for development is an iPhone 8.

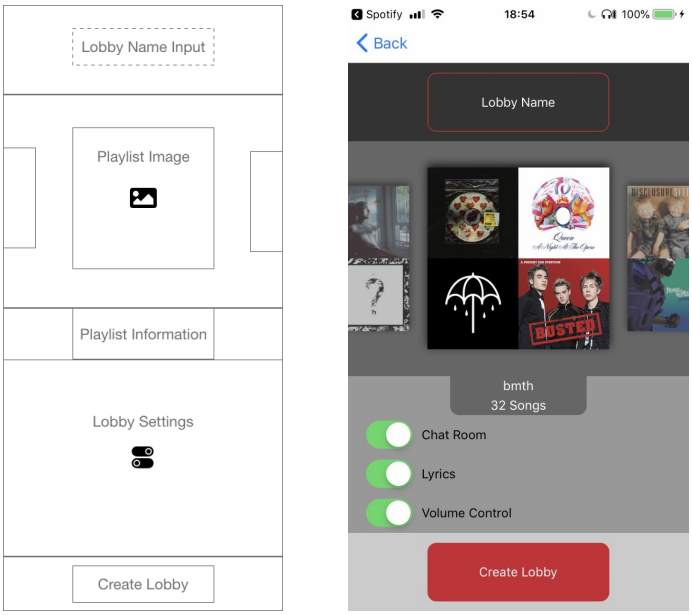
5.2.1 Landing.js



(a) Design (b) Implementation

Figure 5.1: Landing.js (Preliminary UI)

The implementation of the Landing component was the first UI component to be developed, as a result, the reader will notice it has a different colour scheme to the others. As per the designed wireframe, the implemented component serves only 2 functions: Joining a lobby and creating a lobby. Users can join a lobby by typing the lobby key into the text input, then pressing the "Join Lobby" button which sends an HTTPS POST /join_lobby request with the unique device id and lobby key as part of the body. QR code support was not implemented at this stage since it only supplements requirement 1a, and was therefore not a requirement on the sprint backlog.

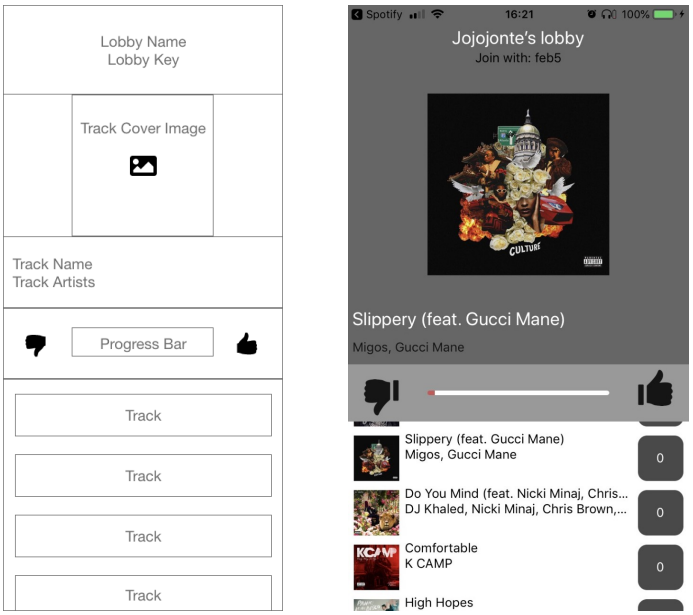


(a) Design (b) Implementation

Figure 5.2: CreateLobby.js (Preliminary UI)

5.2.2 CreateLobby.js

The CreateLobby component (see figure 5.2) was implemented strictly according to the wireframe. Underneath the carousel of playlists, implemented using the react-native-carousel package, is a series of buttons that allows the user to connect the application to their Spotify account, and retrieve a list of their playlists using the /get_playlists endpoint. This endpoint invocation is part of a callback passed as a parameter to the native bridging module getAccessToken method, since a non-expired access token is required to access such data. Once the user completes these actions, the buttons are replaced with the carousel as shown. Users are able to swipe from left to right to browse their playlist library, and the playlist at the forefront of the carousel when the lobby is created is the base playlist. When the user presses the "Create Lobby" button, the /make_playlist endpoint is invoked, with all of the following in the request body: Device id, lobby name, base playlist, and the status of the 3 switches. The switches are either set to true or false, and give the host user the ability to enable and disable the chat room, lyrics being displayed, and volume control. It is important to note that at this point during the development lifecycle none of these had been implemented, only the ability to enable and disable them.



(a) Design (b) Implementation

Figure 5.3: HostLobby.js and InLobby.js (Preliminary UI)

5.2.3 HostLobby.js and InLobby.js

The implementation of HostLobby.js and InLobby.js was perhaps the most complex of all the UI components. This was primarily due to the private lifecycle methods that were part of the components, particularly HostLobby.js. HostLobby.js was implemented first and then it only took minor tweaks and the removal of certain host-device-only methods in order to have implemented InLobby.js too.

During the implementation of HostLobby.js, some extra components were created, such as the track component and the progress bar component. This was done in order to generalise the implementation process and thus increase the usability and consistency, since each of these components would be necessary in both HostLobby.js and InLobby.js.

The track component contains 4 key features, show by figure 5.4: Cover image, track name, track artists and the number of votes for the track. Each of these is passed in to the track as a property, alongside the selected state. The selected state is a state variable whose value is dependent on whether or not the track has been selected by the user. If so, the track will provide some form of feedback in the form of a change in colour scheme. In the preliminary UI, once a track has been voted for it is selected which makes the background colour of the track component a little lighter, so that the user knows that their vote has gone through and they can easily see what they have voted for.

While the progress bar (see figure 5.1) may seem like the much more simple component at the surface, the underlying mechanics behind it are more complex than that of the track component. The progress bar requires the following properties to be passed in to it:

Enabled

A simple boolean that dictates whether or not the progress bar will be visible. The only need for this is that when the host user joins the lobby they have just created, they need to choose a track to play, and there is no need for a progress bar until a track is being played.

Total_time

An integer representation of the length of the track that is currently playing, measured in milliseconds. The progress bar needs to be able to know the length of the track so that it can calculate how much the bar should progress for each factor.

Factor

The "frequency" of the progress bar, the factor represents how many times the progress bar should update over the course of the track. With the time, factor and the pixel length of the progress bar (defined as part of the style, not a specific property), it is possible to very accurately update the progress bar by $\frac{\text{pixel length}}{\text{factor}}$ every $\frac{\text{time}}{\text{factor}}$ milliseconds. By default, the factor is set to 500, so that the progress bar is updated 500 times per track, making it seem like a continuous animated process.

Time_invoked

This property is required so that users who join a lobby after the start of a track see the same progress bar as every other user in the lobby. time_invoked is a property of the current state of the lobby, where it represents the time at which the host device began the playback of the track. It is stored as part of the lobby item in the database and is sent to each user through the "next" route of the WebSocket API or via the /join_lobby endpoint of the REST API. The length of the progress bar at any given time is given by equation 5.1, which is calculated every time the progress bar updates.

$$\frac{\text{Pixel\_length} \times (\text{Current\_time} - \text{Time\_invoked})}{\text{Total\_time}} \tag{5.1}$$

5.2.4 ChatRoom.js

The preliminary UI was implemented before the decision was made to remove the chat room from the requirements. The chat room UI was placed on the

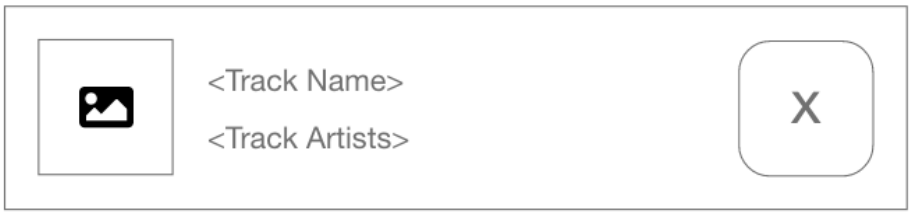
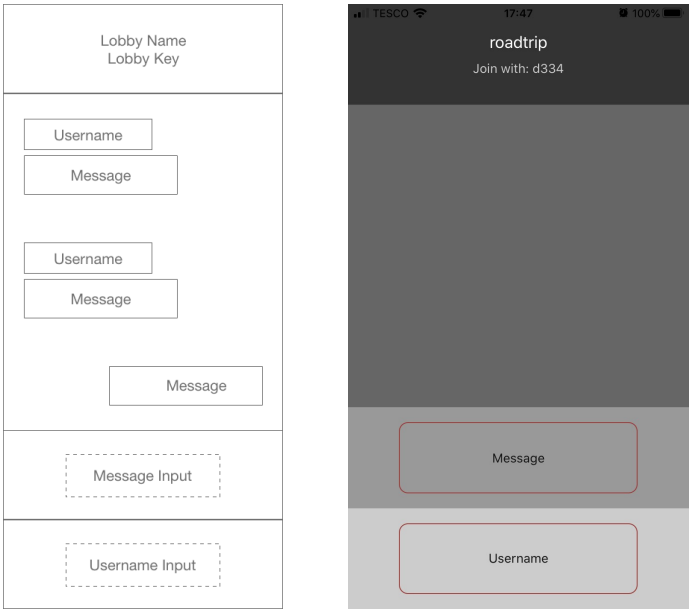


Figure 5.4: Track.js

Figure 5.5: ProgressBar.js



(a) Design (b) Implementation

Figure 5.6: ChatRoom.js (Preliminary UI)

sprint backlog and the it was still implemented as part of the preliminary UI. The messages themselves were not implemented since it would have made more sense to develop the UI support for messages once it was possible to send and receive them.

5.3 Native Bridging Module

As shown in table 5.1, the implementation of the native bridging module took much longer than expected. This is partly due to time that was allocated for learning Objective-C, but namely since the machine that was used to build the application on to the device took approximately 15 minutes to do so. Meaning that each time the bundle server (which is required to build react-native applications) needs to be restarted, it takes 15 minutes. The bundle server needs to be restarted when any changes to the module need to be made. This, when combined with the fact that the language is unknown to the developer, made it a very timely process. Nonetheless, each of the designed methods were implemented successfully.

The first task was to create a module that could be accessed by the client react native application, utilising the NativeModules built-in library. This meant writing a simple Objective-C class and header file, then exporting a simple void method that outputs a string to stdout, as a macro. Once done, the method can be invoked, passing in an anonymous callback function, as this is the recommended way to receive data from a natively-written method. It's an unorthodox form of integration, and for the purpose of clarity, code examples are demonstrated by figure 5.7.

Once the integration between these two fundamental components had been successfully established, it was important to establish communication between the module and the Spotify SDK, so that development on the authorisation process could commence. As per the Spotify SDK "Getting Started guide" [7], the Spotify SDK framework was installed as part of the XCode project. Then, the Spotify framework could be accessed simply by including the classes required.

Figure 5.8 shows the OAuth2 "authorisation code" flow, obtained from the Spotify Web API documentation [12]. The authorisation code flow was chosen over other authorisation flows such as "client credentials" and "implicit grants", since it accommodates both the refreshing of access tokens and the ability to access user credentials. Refreshing access tokens means users do not need to be prompted to grant access to the project application each session (thereby skipping step 1), and accessing user credentials is important to acquire the base playlist. Step 1 of figure 5.8 is done by the Spotify SDK and does not require any authorisation endpoint hosted on the application's backend. Step 2 invokes the /swap endpoint, which will consequently call the relevant lambda function and from there the Spotify Accounts Service will return access and refresh tokens. The backend API will act as a proxy throughout the authorisation process. However, the backend API had not been developed at this stage, and so a node.js script was being locally hosted which used the express library to handle HTTP requests. Once the /swap endpoint was functional, the module was able to store the access and refresh tokens as session attributes, and then when session methods such as playURI were invoked (to play a track), the access token would be given as part of the request to the Spotify application. The /refresh endpoint was also implemented locally which meant if a session had expired

```objc
- (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;
  }
}
```

(a) Native Implementation

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

(b) Exporting the method as a macro

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

(c) Invocation from react native

Figure 5.7: Integration with the native bridging module

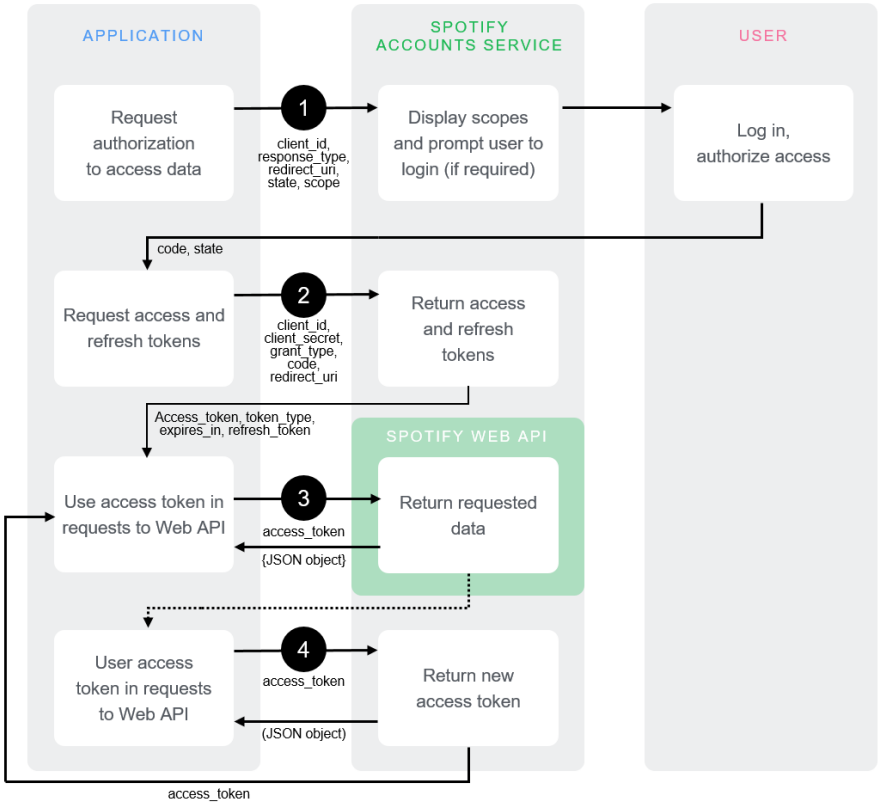


Figure 5.8: Spotify Authorisation Code Flow

(as it does after 1 hour), then the module could retrieve a new access token using the refresh token. It is clear that going forward, in order to implement a comprehensively facilitate authorisation, the backend API hosted on AWS would need to persist the refresh token for each host user.

The implementation procedure was timely and there were many bumps in the road regarding the development of the native bridging module. Other than those already mentioned, the specific implementation difficulties included Objective-C's lazy function evaluation, and the fragility of the Spotify SDK appRemote connection.

5.3.1 Lazy Function Evaluation

If a compile-time constant is to be returned by a function, then when it is called the function may return the constant before the body has been executed. This feature of the Objective-C compiler, LLVM, renders the callbacks of such functions superfluous. The callback can be invoked before the method has

finished executing, meaning any parameters the callback may require could be undefined. This problem was first encountered when the authorisation method of the module was called which returned a "YES" (Objective-C equivalent of a logical true). As a result, the callback parameters parameters would always be null, making it impossible to know if the authorisation had succeeded. More importantly, it had meant that any consequent body of code, that may rely on the successful completion of authorisation, would execute before any such authorisation had taken place.

In order to fix this problem, semaphores were utilised to force the thread to wait until the semaphore had been signalled before proceeding through the block of function code. The semaphore would be instantiated at the beginning of the function, and once the evaluation had concluded, it would be signalled. Placed just before the return statement of the function, the dispatch_semaphore_wait method told the thread to wait for the signal before returning. Once the function returned, the callback that had been passed into the invocation in the client application could be evaluated with the correctly assigned parameters.

5.3.2 Fragility of the Spotify appRemote connection

The `appRemote` is an object that provides an assortment of APIs which can be used to easily interact with Spotify application features, such as controlling playback or retrieving user information. It has 2 possible states, connected and unconnected, to the Spotify application. The `appRemote` can only be connected if an unexpired access token is stored as part of the session. If the user is not using the application, the `appRemote` must be disconnected, and if they are to reload the application then a new connection attempt must be made. In order to connect the `appRemote`, there must be music playing on the device, due to iOS security restrictions. It is inconvenient for the user to be required to have music playing in the background while the project application connects to the Spotify application.

To avoid such inconvenience, during the authorisation process the `appRemote` will attempt to connect. If music is playing, it will connect, otherwise it will not. However, when the host user first joins the lobby they will be asked to initiate the lobby music playback by selecting a track from the set of recommendations. At this point, if the `appRemote` is connected, it will simply call the playURI method part of it's API. Conversely, if it is not, the module will invoke the authorizeAndPlayURI method. This method will attempt to authorise the user again, and when the app-switch is made to the Spotify application for authorisation, it will play the requested track before attempting to connect the `appRemote`.

5.4 Database

The implementation of the database was simple, using the online AWS Dynamo interface, all of the designed tables were implemented and no schema was re-

quired as is the nature of DynamoDB, a schema-less service. The tables are all empty by default and will be populated eventually by the lambda functions. The AWS-SDK which is a pre-installed package as part of the lambda system environment gives lambda functions easy access to the DynamoDB API. Using this, it is possible to insert typed JavaScript objects in to the tables, figure 5.9 demonstrates example node.js code that would be used in a lambda function to retrieve the stored recommendations for the lobby that a user is in. In this example, the dynamo object represents the API that is provided as part of the AWS-SDK.

```
/* (1) Get lobby_key from uid */

let lobby_key_res = await dynamo.getItem({
    TableName: "device",
    Key: {
        "device_id": {"S": req.uid}
    },
    AttributesToGet: ["lobby_key", "user_weighting"],
}).promise();

let lobby_key = lobby_key_res.Item.lobby_key.S;
let user_weighting = lobby_key_res.Item.user_weighting.N;

/* (2) Get recommendations list fromm lobby item */

let recommendation_list_res = await dynamo.getItem({
    TableName: "lobby",
    Key: {
        "lobby_key": {"S": lobby_key}
    },
    AttributesToGet: ["recommendations"]
}).promise();

let recommendations = recommendation_list_res.Item.recommendations;
```

Figure 5.9: Retrieving recommendations from the database

Similar to that of certain Objective-C methods, some DynamoDB API interactions such as updateitem and getitem evaluate to null before the action is completed, which is updating and getting an item respectively. This is not much of an issue when using the updateitem method, where the action is still completed and what the invocation evaluates to is not important. However, this can and did lead to confusion when using methods such as getitem, where the purpose of the method is that it returns the attributes of a specific item upon invocation. This is due to the fact that database queries are mostly asynchronous. As a result, and discovered through experimentation, the query can be converted to a promise, then the compiler can be asked to "await" the fulfil-

ment of said promise. Using the JavaScript ES6 async/await syntax and with the whole process wrapped in a try-catch statement, if an error is thrown after querying the database, the lambda function will respond with a HTTP 500 (Internal server error) status code. The process of converting asynchronous queries into promises and awaiting their fulfilment is demonstrated twice in figure 5.9.

5.5 REST API

As mentioned in section 5.3, the rest API was initially implemented as a single node.js file that was hosted locally. This was done in order to support the development of the authorisation process for the native bridging module, and to implement basic resources such as the /make_lobby resource, which enabled the preliminary UI to be developed in full. The single-file node.js backend had functioning endpoints like /make_lobby and /join_lobby, endpoints which were designed to require database support. Instead, the early single-file implementation of the backend utilised lobby and user objects, demonstrated by the class diagram shown in figure 5.10.

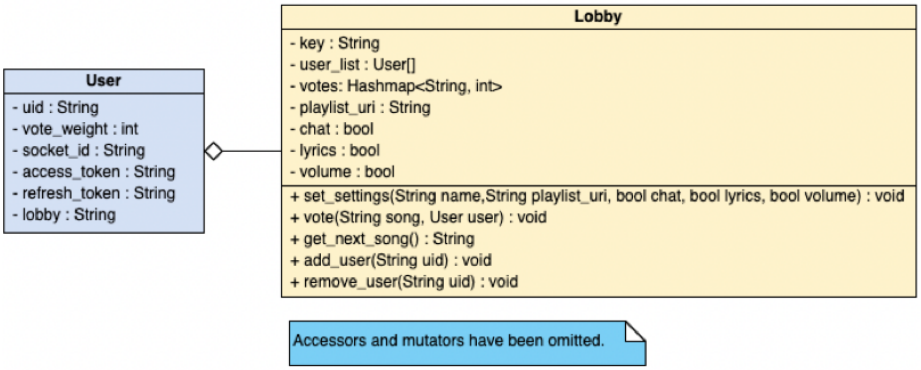


Figure 5.10: Class Diagram of User and Lobby

Once the Native Bridging Module and the preliminary UI had been developed the rest API could be migrated to the AWS servers. Doing so was very simple for authorisation resources, since the underlying functionality remained mostly the same; Resources such as /swap and /refresh needed very little tweaking in order to function correctly having been migrated. Conversely, other resources, like /join_lobby or /make_lobby, were complicated to migrate since instead of using lobby and user objects, the database was to be used. Consequently, the migration of authorisation resources was performed immediately, but the development of resources outside the domain of authorisation was timely. In spite of this, it is undoubtedly the more able implementation both in terms of efficiency and persistence.

Firstly, a new API was created as part of the API Gateway service provided by AWS. In doing so, a Uniform Resource Locator (URL) is allocated to the API.

Following this, resources can be created, these represent are the endpoints of the API, so the /swap and /refresh endpoints at this stage. Alongside the resources, the API Gateway interface allows API engineers to associate a resource with a set of HTTP methods. In this case, OAuth2 dictates that POST is the required method for both /swap and /refresh. Then, a node.js lambda function is created in the AWS Lambda interface, making up the code that is to be executed upon successful invocation of the corresponding endpoint. As a result, both swap and refresh lambda functions were created and linked to the API resources using proxy integration, demonstrated by figure 5.11. With this, the corresponding lambda functions will execute once a POST /swap or POST /refresh request is sent to the API.

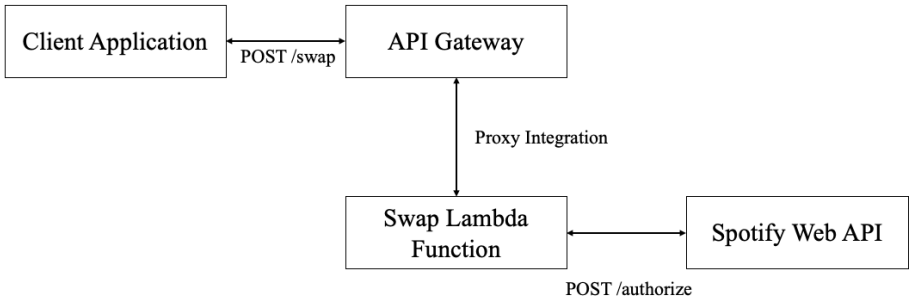


Figure 5.11: Architecture of REST API endpoint invocation

However, the early single-file API code was dependent axios and express, both node packages. When using a combination of lambda functions, they can only interact with each other by using the API they themselves compose. That is to say, they're completely modularised, and each lambda function needs its own node_modules folder if it requires a collection of node packages. In order to perform the installation of a node package to lambda function, the simplest method is to write the lambda function code locally as part of an index.js file contained within its own directory. Then, initiate node package manager, install the required packages to the node_modules directory, zip the entire lambda function directory and upload it to the online AWS Lambda interface. This was the method employed in the implementation of all the lambda functions. All of the endpoints of the API were developed alongside their associated lambda functions, but not all of the lambda functions were fully implemented, since they may depend on functionality that is yet to be developed. For example, the /get_recommendations endpoint cannot be fully implemented until both the artist and genre classification models are designed, trained and deployed.

5.5.1 Lambda Scalability

High modularisation increases readability of the API as a whole but drastically increases overhead too, since for each endpoint invocation the dependent node

modules need to be loaded into memory. As a result, if a sequence of endpoints need to be called in quick succession, each reliant on the previous invocation, then the latency accumulates. To decrease this overhead, the API could employ a monolithic structure which would mean overhead is significantly reduced. However, while the migration of the API to a monolithic structure would be relatively simple, it would cost much more to have a constantly running AWS S3 instance act as the server, and low latency is not a specific requirement. Moreover, requirement 3a specifies that the API should be scalable, and it could be argued that AWS Lambda is more scalable than its monolithic counterpart. Since the code that composes lambda functions is stateless, lambda can start as many copies of a function as needed, and dynamically allocate capacity to match the rate of incoming events [14]. Ultimately, the backend API sacrifices latency for high scalability.

5.5.2 Lyrics Support

Requirement 3c specifies that the lyrics for the currently playing track should be displayed as the track is being played, if the lobby setting is enabled. When it came to the implementation of this functionality, the Spotify Web API could not be utilised, since there is not a /get_lyrics resource of any sort. Spotify use the Genius API [15] to obtain their lyrics data. Therefore, a /get_lyrics endpoint was created as part of the REST API and a lyrics lambda function was made. The idea was that the lambda function would take the Spotify track id and output the lyrics for the track. This seemed simple enough, but it turns out the Genius API do not offer a lyrics accessor resource either, and after further inspection, it became clear that this was for copyright reasons. Instead of lyrics, however, the Genius API do offer "annotations" for tracks. These annotations are user-written descriptions of the track, which could include the meaning behind the track, or how and when was written/made. While this is a completely different feature and certainly does not satisfy requirement 3c, it does offer functionality and information that the user can interact with, for each track, which is essentially the purpose of the lyrics feature. Consequently, as previously described in section 5.1.1, lyrics support was replaced by track information in requirement 2e, and requirement 3c was completely removed from the project, since there is no need to synchronise the track information with its playback. This process is in alignment with the contingency plan highlighted in section 3.4.3, where requirements may be amended or removed in the case of a legal, social or ethical issue.

The Genius API does not offer a service that maps a Spotify track id to a Genius track id, but does store the Spotify id as part an external URL within the track object of some tracks. The /search endpoint had to be utilised in order to find Spotify tracks within the Genius API. The Spotify id is used to determine the track name (using the Spotify Web API), and then the track name is used to attempt to find the track as stored on the Genius API, which can then be used to acquire the annotations. As shown in figure 5.12, a small adaptation was made to the UI so that the user could double tap the album

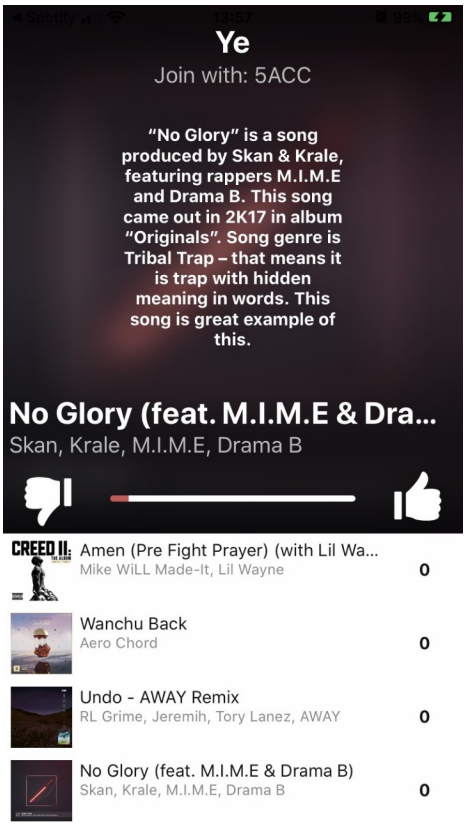


Figure 5.12: Track Information Support

cover of the currently playing track and it would be replaced with the fetched annotation. This worked for most tracks, but there are inconsistencies in the names of tracks as stored on the different APIs, meaning that it is often the case that the annotations cannot be found, and since they're user-written, the actual annotations could be inaccurate, profane, meaningless, etc. For this reason, the feature was disabled but the UI adaptation remained since it offered the possibility for users to double-tap to obtain the lobby QR code, once this functionality had been developed.

5.6 WebSocket API

Figure 5.13 shows the architecture of the "next" route, where the next track has begun playing on the host device, so it will notify the WebSocket API utilising the next route. Upon receipt of this message, the API will invoke the "next" lambda function, which will identify all users in the same lobby as the host device, obtain their connection ids, then send a "next" message through

all of these connections. The message will invoke the event handler as part of the InLobby.js component that manages incoming WebSocket messages, and prompt devices to send a /get_next_track request to the REST API, and thus acquire details of the track that has recently begun playback on the host device.

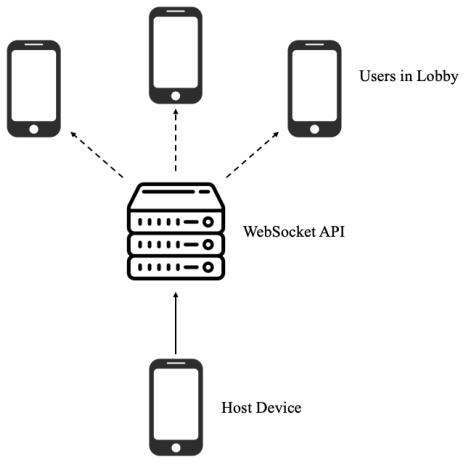


Figure 5.13: Architecture of the next route

The "vote" route takes the same architecture as the "next" route, except that any device can send a message through the vote route, it is not exclusive to the host device, since any user can vote. Figure 5.14 shows a snippet of code which is part of the vote lambda function. This code takes a list of users connected to the API which are part of the same lobby as the user who voted, then sends an updated set of votes according to the database to each user apart from the user who voted. The function does not send the vote data back to the user who invoked it, because lambda functions can at times have relatively high latency, and it would be perceived by the user that their vote is taking a lot of time to register with the system. Instead, when a user votes, a client-side variable is assigned the index of the recommendation which was voted for, so that the vote number for that recommendation can be incremented by the user's vote weight almost instantly. Then, if said user were to receive another message from the API that a different user had voted for another track, the client-side component would disregard the increment made by itself. Instead, it would use the full set of votes as given by the message since it's a more up-to-date set of votes.

The code shown in 5.14 accommodates HTTP 410 Gone Client errors, in the event that a user unexpectedly had their connection interrupted. In such a case, the item will be deleted and the connection forgotten. This is an unlikely scenario since the $disconnect route should handle most connection interruptions. More importantly, the updated vote objects are sent to each device (apart from the host device) simultaneously, through the utilisation of the Promise.all method, which takes an array of promises and attempts to fulfil them concur-

```
const postCalls = connected_users.map(async (item) => {
    try {
        console.log(item);
        if (item.device_id.S != req.uid) {
            await apigwManagementApi.postToConnection({
                ConnectionId: item.connection_id.S,
                Data: JSON.stringify({
                    action: "vote",
                    body: vote_array
                })
            }).promise();
        }

    } catch (e) {
        if (e.statusCode === 410) {
            // HTTP ERROR 410: Gone Client
            // Delete connection
            await dynamo.deleteItem({
                TableName: "lobby-connection",
                Key: {
                    "lobby_key": {"S": lobby_key},
                    "connection_id": {"S": item.conneciton_id.S}
                }
            }).promise();
        } else {
            throw e;
        }
    }
});

await Promise.all(postCalls);
```

Figure 5.14: Node.js implementation of vote route

rently.

5.7 Recommendation System

5.7.1 Genre Classification Model

Data Pre-processing

The 1 million playlist dataset is no longer public, and no publicly available datasets of playlists contain un-hashed Spotify ids of tracks, playlists, artists, etc. Therefore, it is infeasible to build a model trained on public datasets that can interact with the Spotify Web API as is required.

A Python 3.4 script was written that utilised the Spotipy package [16] in order to interface with the Spotify Web API, to collect the data required to build the genre classification model. The data required to build the genre classification model was simple: A CSV (Comma Separated Value) file composed of 2 fields, playlist_name and genre. The initial approach was to take one user, get that users's playlists, determine the genre of each playlist based on its composing tracks, and write this data to the CSV. Then, do the same for all that user's

followers, and their followers, etc. This breadth-first-search would very quickly build a diverse dataset which could be used to train the model adequately. Unfortunately, the Spotify Web API does not provide user follower information. Seemingly, the Spotify Web API contains no resource that accommodates easy propagation or traversal. Moreover, there was no clear way to determine the genre of the playlist based off its composing tracks, since the Spotify Web API does not provide any resource to determine the genre of a specific track.

The /search endpoint provided by the Spotify Web API helps users find the tracks, albums, artists or playlists that they're looking for. It takes a string as the query and can return a collection of playlists if the playlist flag is set. It can also be used to obtain public Spotify data. For example, a script can iterate through a large dataset of words, use /search with each word and collect playlist data, solving issue 1.

Issue 2 is that there is no defined way to determine a playlists genre. Instead, a heuristic was developed to approximate it. While Spotify does not store genre information for each track (or at least does not allow public access to such data), they do make each artist's main genres available. For each track that composes a playlist, the main artist's genres are accumulated. The most common genre among the main artist of the tracks of the playlist is the perceived genre of the playlist. This approach has two main potential pitfalls: The track may not belong to any of the main artist's main genres, and the main artist may contribute less to the track than some of the other artists as part of the same track. However, these are edge-cases and likely result in negligible approximation error, particularly on large playlists. Moreover, executing the heuristic on a playlist, but accumulating the main genres of each artist changes the time complexity from $O(n \times g)$ to $O(n \times g \times a)$ (assuming all Spotify Web API interactions takes $O(1)$ time), where n, g, a represent the number of tracks, genres and artists, respectively. Though both take polynomial time, a pragmatic solution would employ the former, since this is a heuristic that will likely be executed on hundreds of thousands of playlists. Also, acquiring the main genres of an artist requires an API invocation, and while this is assumed to take $O(1)$ time, API interactions will definitely act as a bottleneck for the scraping process. Therefore, it is argued by the developer that the former is a relatively pragmatic heuristic, and can be utilised to approximate the class labels for each example in the genre classification model. Combining these two solutions results in a script demonstrated in pseudo-code by by algorithms 1 and 2. In these algorithms, the search and main_genres methods represent abstractions of /search and /artist endpoint invocations. The words array is initialised to contain approximately 466000 English words [17].

The script was built to save the scraped data every 100 words in a new CSV, at which point the access token used to interact with the Spotify Web API would be refreshed using the /refresh endpoint. The CSVs could then be concatenated using another Python script to produce one large dataset containing all the data that the script has scraped. After 5 hours of the script running, there were 285 CSVs, meaning 28500 words had been used to search for playlists, and a total of 318379 playlists had been scraped, and their genres approximated. On average,

Algorithm 1 Traversal

```
words ← [...]
procedure TRAVERSE
    data ← []
    for word in words do
        playlists ← search(word)
        for playlist in playlists do
            genre ← GET_GENRE(playlist)
            data.append((playlist.name, genre))
        end for
    end for
    return data
end procedure
```

Algorithm 2 Genre approximation

```
procedure GET_GENRE(playlist)
    acc ← {}
    max_genre ← ϵ
    tracks ← playlist.tracks
    for track in tracks do
        artist_id ← track.artists[0].id
        genres ← main_genres(artist_id)
        for genre in genres do
            acc[genre] ← acc[genre] + 1
        end for
    end for
    for genre in acc.keys do
        if acc[genre] > acc[max_genre] then
            max_genre ← genre
        end if
    end for
    return max_genre
end procedure
```

each word yields approximately 11 playlists.

It was evident, however, that there were limitations to using a large list of English words. For one, many non-English words will be Out-Of-Vocabulary (OOV) for the model, reducing the availability of the application. Also, it was evident that shorter, more general words, or even sub-words like "ing" produced plenty more results than large specific words such as "nonsubstitutionally". The latter is an example of a word that will not find itself belonging to many playlist names. Furthermore, there are no spelling errors in this dataset of English words, and playlist names are written by people and therefore subject to error, if the model were to be trained using only correctly spelled words, then a large proportion of lobby names names would be OOV. In order to counteract these issues, the script was adapted to generate random permutations of 3 and 4-letter sub-words and use these as the search query. These random sub-words have no language and will collect playlist names written in a large variety of languages, and since these random permutations have no obligation to be real words, playlist names with spelling errors will also be found. In addition, the playlist names from the #nowplaying music dataset [18] were used to create another adapted script where instead of using English words or random sub-words to search, random playlist names were used, gathering domain-specific data. Then, all 3 of these datasets could be combined to build a much larger dataset, with both general (random sub-words and English words) and domain-specific (using real playlist names) data, catering for correctly and incorrectly spelt words, English and non-English.

With these various datasets processed, there was a total of 1700 different class labels (genres). However, there are only 126 genres that can be used as seeds in the /recommendations endpoint as part of the Spotify Web API, which turns the inferences into tracks. Therefore, in order to have a usable model, the number of classes in the dataset needs to be reduced down to 126, through the clustering of classes. Cosine similarity is an algorithm that "gives a useful measure of how similar two documents are likely to be in terms of their subject matter" [ref from cos similarity wikipedia]. In this case, 2 class label strings would be converted into vectors so that the cosine similarity between the 2 strings could be calculated.

In essence, genres that share lots of similarity in plaintext are assumed to be of similar genre generally. For example, the genre "Swedish hip hop" shares most plaintext similarity with the more general cluster "hip-hop" than it does with "folk", "classical", etc. As a result, after execution of algorithm 3, all examples in the dataset who have the class "Swedish hip hop" will now have the class "hip-hop". This new class can be used in tandem with the Spotify Web API /recommendations endpoint to obtain a set of tracks belonging to that genre, in this case, "hip-hop". The author is aware that using a string comparison mechanism to cluster the classes is relatively naïve. It was immediately evident following the execution of algorithm 3 that there are some erroneous cases; "grime", a British rap genre, shared most similarity with "anime", a genre primarily composed of Japanese cartoon soundtracks. A more sophisticated approach could attempt to mitigate these blatantly erroneous cases, maybe by

Algorithm 3 Cosine Similarity Clustering

```
data ← read_csv(concat.csv)
usable ← {g_1, g_2, ..., g_1700}
procedure CLUSTER(data, usable)
    labels ← data.labels
    map ← Map()
    for label in labels do
        max_cos ← 0
        max_cluster ← ϵ
        for label2 in usable do
            cos ← cosine_similarity(label, label2)
            if cos > max_cos then
                max_cos ← cos
                max_cluster ← label2
            end if
        end for
        if max_cos ≥ 0.5 then
            map[label] ← label2
        end if
    end for
    data ← data[genre in map.keys]
    return data.apply(map)
end procedure
```

learning embeddings for each or through the utilisation of an autoencoder, and then clustering using k-means. However, the time constraint at this point of development was large, and so the decision was made by the author to employ the cosine similarity clustering method with a catch. If the cosine similarity is less than 0.5 then any playlist belonging to said genre shows negligible sign of plaintext similarity with the usable set of genres, and thus it will not be used to train the model. This condition helps to ensure that the model is not trained on false conclusions.

Model

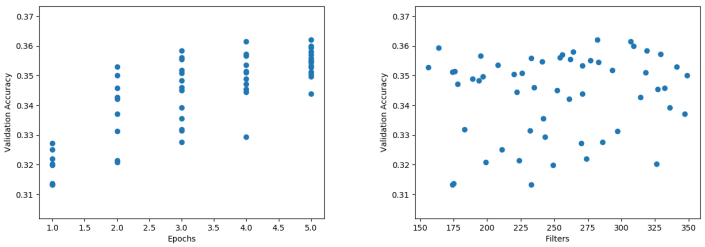
Using the general architecture shown in figure 4.4, random hyperparameter tuning was performed to establish perceivably optimum hyperparameters. Since training the model on the total dataset can take upwards of an hour per epoch on the development machine, a 10% sample of the dataset was taken for hyperparameter tuning. The success of a set of hyperparameters is determined by the validation accuracy obtained following the model being trained with said set of hyperparameters. The tuning parameters and ranges are as follows:

1. **Epochs**: 1-5
2. **Number of Convolutional Filters**: 150-350
3. **Convolutional Kernel Size**: 5-50
4. **Dropout Percentage**: 0-40%
5. **Embedding Dimensionality**: 50-150

The tuning job completed 60 different training jobs meaning 60 distinct sets of hyperparameters. Initial model weights were randomised using the constant random seed of 42 to mitigate the skewing of results that would occur if weights were initialised differently each iteration of the job. The results for each hyperparameter can be seen in figure 5.15. Some models had the same validation accuracy, so it may seem that there less than 60 models had been trained.

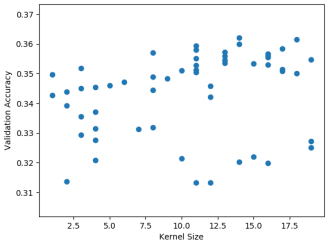
As is evident, the only hyperparameter that has any form of correlation with validation accuracy in spite of the random nature of the other parameters, is the number of epochs. It is clear to see that the validation accuracy converges to a global maxima of approximately 36% as the number of epochs is increased. The quasi-absence of correlation between the other parameters and validation accuracy could therefore be explained by the massive skew incurred through variance of the number of epochs. In order to ascertain this and thereby gain insight as to what parameters increase validation accuracy and how, another tuning job was initiated. This second job trained 30 models, with the same random hyperparameters highlighted earlier, using the same ranges, with the only exception of the number of epochs (which was set to a constant 5). The results of this job are shown in figure 5.16.

Unfortunately, the second hyperparameter tuning job provided no further insight in to the question of whether or not there exists a hyperparameter that

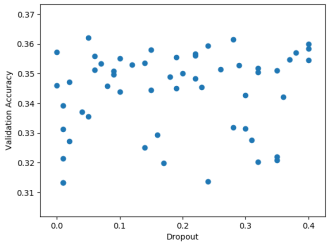


(a) Number of Epochs

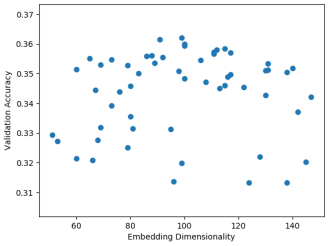
(b) Number of Filters



(c) Kernel Size



(d) Dropout

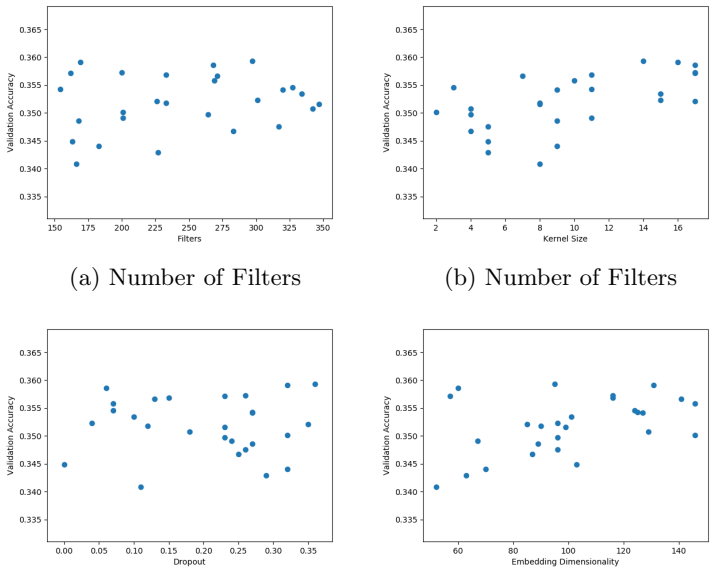


(e) Embedding Dimensionality

Figure 5.15: First hyperparameter tuning job

is verifiably more influential than the others. It can be argued that this may be due to the influence that each parameter has on the effect of the others. This can be seen as the case where any two or more models share the same value for a parameter, yet attain different values for validation accuracy.

The third and final hyperparameter tuning job was initiated over 60 models in order to determine the parameters that produced a global maxima when 3 epochs were used to train the model, again on a 10% sample. These hyperparameters would finally be used to train an extrapolated model, where the entire dataset would be used, on 5 epochs. The hyperparameter tuning job only used 3 epochs in the interest of saving time. The model maximum validation accuracy was attained by a model consisting of 295 convolutional filters, a kernel size of 17, a 24% dropout and an embedding dimensionality, as seen in table 5.2.



(a) Number of Filters

(b) Number of Filters

(c) Dropout

(d) Embedding Dimensionality

Figure 5.16: Second hyperparameter tuning job (No. Epochs controlled)

	Epochs	No. Filters	Kernel Size	Dropout	Embedding Dimensionality	Validation Accuracy
Min	3	157	6	40%	71	34.7%
Max	3	295	17	24%	130	38.1%

Table 5.2: Results of the third hyperparameter tuning job

Figure 5.17 demonstrates the validation accuracy after a number of training epochs for 2 models, 1 and 2. The 1st model was trained on 80% of the total concatenated dataset and validated against the remaining 20%. Once each epoch concluded a callback method was invoked which measured the validation accuracy, thus removing the need to create 5 different models all trained on a different number of epochs to produce the same result. The validation accuracy after 5 epochs was 51%, a substantial increase on the model trained with the same parameters using 10% of the dataset. It is evident that the size of the dataset has a profound impact on the accuracy of this model, probably since it is fundamentally a Natural Language Processing (NLP) model, which requires many words to perform well. The number of words contained within the dataset is 214368. It is also clear that the validation accuracy begins to plateau after the second training epoch concludes, and only negligible fluctuation occurs thereafter. To test the effect the volume of data has on the validation accuracy, 50% more data was collected from the scripts in an attempt to produce a new model which is more accurate on unseen data. This new model, "model 2",

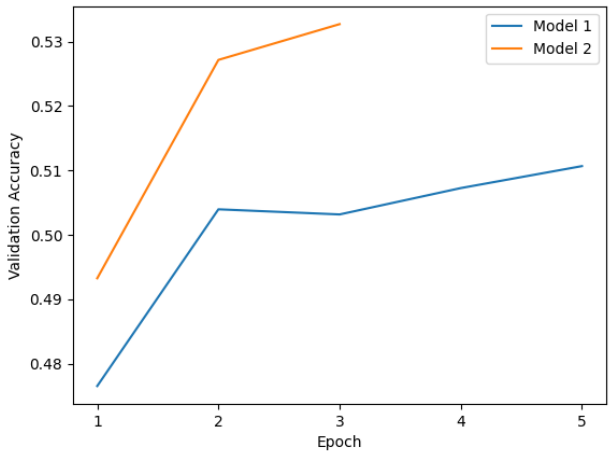


Figure 5.17: Model 1 vs Model 2

was trained on only 3 epochs, since model 1 has shown how validation accuracy varies after the second epoch, and is therefore unnecessary for this test. Model 2 achieved 53.3% validation accuracy; a 2.3% improvement. The model weights and configuration were both saved so that the model could be deployed later on.

While 53.3% may not seem like a large accuracy score for a classification model, it is worth noting that the model is trained on user-data, where it is certainly possible that one input maps to various outputs. For example, in the training data there are plenty of cases of playlists that have been found that have the same name, yet are composed of different tracks that yield a different genre when given to the genre approximation heuristic. Consequently, this helps the model to give high probabilities of classification for a set of genres, represented in the output of the last hidden layer, where the highest 3 values represent the most likely classifications. If each input only mapped to one output, then the model would struggle to learn rankings for each input.

5.7.2 Artist Classification Model

The artist classification model will take a sequence of 3 track features, along with their main artists, and predict the main artist of the next track that the user will listen to. In this case, user-built playlists will be used to mimic sequential listening sessions. After all, the competitors for the 2018 RecSys Challenge used playlist data as the foundation for their models too. While the resultant model will be categorised as an APC model, it is perfectly capable of recommendations, due to the fundamentally sequential nature of the tracks played by the lobby, which can themselves be used to compose a playlist.

Before the model was developed or any data was gathered, the mechanism that determines the input to the model (see section 4.6.2) was first implemented into the /get_recommendations resource as part of the backend REST API. An ordered array of heterogeneous pairs is stored in each lobby item within the lobby table of the database, demonstrated by equation 5.2, and is updated for when any track concludes within the lobby. The pairs are composed of an x_i and a t_i, these represent the track_weighted_ranking and the track_id respectively. The track_id can be used to fetch track data such as features and artists, meaning that only the track_id actually needs to be stored in the database. This data can then be used to provide the inputs to the model.

$$L = [(x_1, t_1), (x_2, t_2), (x_3, t_3)] \tag{5.2}$$

Data Pre-processing

The first challenge was obtaining playlist data, and a similar approach in terms of API traversal to the data-scraping script of algorithm 1 was adopted. The array returned by algorithm 4 is an array of pairs, each pair represents a playlist. The first element of each pair is an array containing the main artist_id for each of the 3 tracks, and the second element is a 3x8 array consisting of the 8 track features for each of the 3 tracks. In algorithm 4, the search and get_track_features methods represent abstractions of the /search and /audio-features endpoints of the Spotify Web API.

With this newly accumulated data, a CSV can be built that contains precisely the following fields: playlist_number, track_number, artist_id, acousticness, danceability, energy, instrumentalness, liveness, loudness, speechiness, valence. Such a CSV was created to easily construct inputs and ground-truths for the model, representing each track as (artist_id, [track_features,]...). For each playlist, the tracks are split into groups of 4, maintaining order within each group. The artist of the final track in each group is seen as the ground truth and the remaining 3 tracks are the input. Using algorithm 4, 27456 playlists amassing 1082322 tracks and 84469 artists were collected after approximately 10 hours of runtime. The most common artist in this dataset is Ed Sheeran, with a frequency of 4891. This implies that a naive classifier would achieve an accuracy of 5.79% by always predicting Ed Sheeran, which can be used as a baseline comparison for the model's validation accuracy.

Model

The model architecture designed in section 4.6.2 was implemented using the keras functional API [19]. The functional API was chosen over the sequential one since the functional API is more able to accommodate multiple inputs.

After executing an initial training job with standard hyperparameters, it was clear that a different training and testing approach to that which is most conventionally adopted was required. Attempting to fit the model by simply passing in the input matrices and ground-truth vector led to the development

Algorithm 4 Pseudo-code data-scraping script for the artist classification model

```
procedure BUILD_DATASET(playlist_names)
    data ← [ ]
    for playlist_name in playlist_names do
        playlists ← search(playlist_name)
        for playlist in playlists do
            track_features ← get_track_features(playlist.tracks)
            artists ← [ ]
            for track in playlist do
                artists.append(track.artist_id)
            end for
            data.append((artists, track_features))
        end for
    end for
    return data
end procedure
```

machine attempting to load 126GB into memory. This, of course, meant that the training job failed as the machine simply could not allocate the capacity of memory requested. To counter this, a Python generator was developed that incrementally determined and loaded into memory the data required for training/testing each batch. The artist ids were encoded as positive integers and one-hot-encoded within the generator for each batch. Consequently, the shape of X changed from (255946, 3, 84478) to (255946, 3, 9), making the training and testing of the model much more space efficient. Conversely, CPU overhead is increased, as the computation for each batch has to be done incrementally, instead of all at once.

The resultant large CPU overhead makes hyperparameter tuning a difficult task. On the entire dataset, it takes approximately 4 hours to train the model for each epoch, and it is unknown how many epochs are required to make the validation accuracy converge. Therefore, a Python script was written to produce small subsets of the entire dataset, in the hope that the results of any hyperparameter tuning job would produce an optimum set of parameters that could be scaled up later to train and test a model using the entire dataset.

With very small subsets of data, the proportion of classes to examples is higher. This is shown by figure 5.18, where the gradient of the graph represents said proportion, which generally decreases with larger datasets. Such behaviour can be attributed to the data-scraping algorithm 4, which becomes more likely to collect already seen artist ids as the total amount of data it has scraped increases. The model will be less equipped to learn the relations between classes (artists) with a higher proportion of classes to examples. Moreover, it is much more likely that unseen artist ids will be given to the model if it is trained on a very small subset of data. Thus, validation accuracy is likely to suffer on very

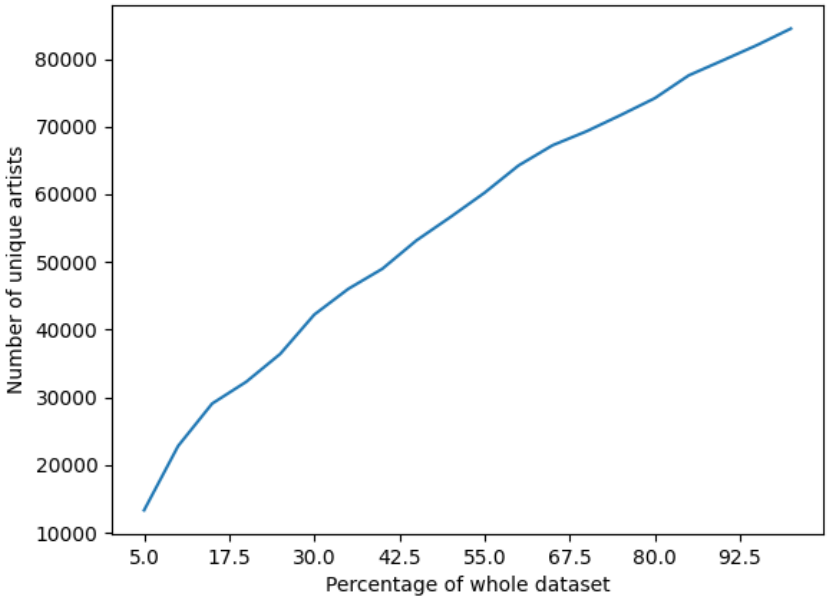


Figure 5.18: Measuring the number of classes composing different sizes of subset

small subsets of the data, and should not be used to scale any hyperparameter tuning results to larger datasets. The greatest gradient of figure 5.18 can be seen preceding the 10% mark, the gradient begins to partially stabilise (though is certainly not linear) thereafter. As a result, a subset of 10% the size of the entire dataset provides a good trade-off between speed and scalability for a hyperparameter tuning job.

Training a model on 10% of the dataset with the same initial parameters costs approximately 10 minutes per epoch. While this is a substantial improvement on 4 hours, a tuning approach similar to that of the genre classification model where 60 models are trained with 3 epochs would take approximately 30 hours, making it largely impractical. Instead, the approach adopted by the artist classification model takes the 10% dataset and assumes that the effect of each hyperparameter is independent from one another. This approach allows each hyperparameter to be tuned individually, one at a time, in the hope that the perceived optimum hyperparameters could be implemented on a model which is trained and tested on the entire dataset.

Epochs

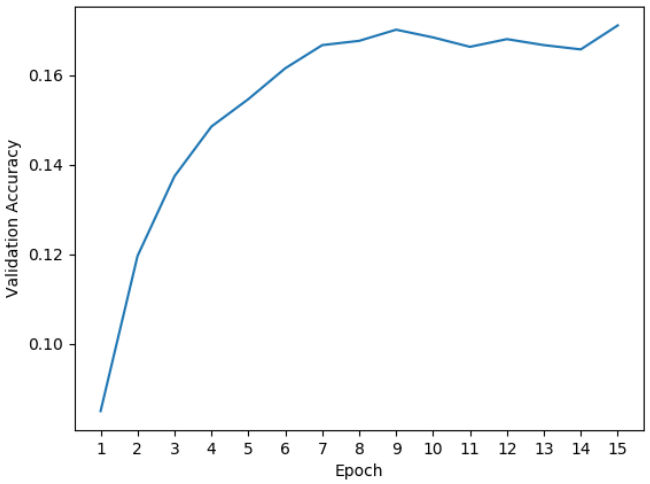


Figure 5.19: Validation Accuracy vs Number of Epochs

Demonstrated by figure 5.19, the validation accuracy begins to converge after approximately 7 training epochs. After which, it fluctuates negligibly. Therefore, the final model will be trained using 7 epochs. However, the following hyperparameters will be tuned using only 1 epoch, since the validation accuracy attained each iteration is only useful when compared to the other iterations.

Embedding Dimensionality

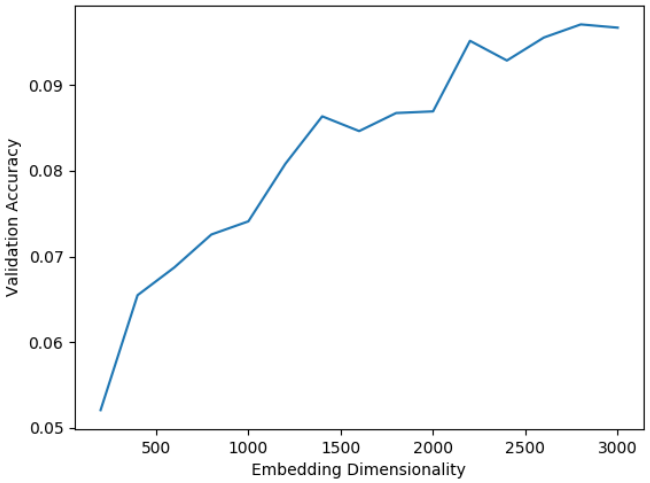


Figure 5.20: Validation Accuracy vs Embedding Dimensionality

Embedding dimensionality represents the dimension of the space output by the embedding layer, and it is evident from figure 5.20 that it shares a strong positive correlation with validation accuracy. The minimum validation accuracy was attained by the model with an embedding dimensionality of 200, which took 43 seconds to train and achieved a validation accuracy of 5.21%. The most accurate of all the 15 trained models was the model with an embedding dimensionality of 2800, which achieved 9.71% and took 427 seconds to train. Both models were trained using a single epoch. While the change in validation accuracy is substantial, so is the difference in time taken to train each model. The final model may not be able to employ an embedding layer with such high dimensionality if it means the training time is impractical. With time in mind, a compromise will be made and an embedding dimensionality of 2250 will be used in the final model.

RNN Dimensionality

The RNN Dimensionality parameter describes the dimension of the output space of the recurrent layer. From figure 5.21 it is clear that there is some positive correlation between RNN Dimensionality and validation accuracy, though it is relatively unstable and the difference in validation accuracy between the minimum and the maximum is only approximately 2%. 15 different models were trained, each consisting of a single epoch. The maximum validation accuracy was attained by the model with an RNN Dimensionality of 360, and the minimum with 120. The latter took 217 seconds to train, and the former took 214

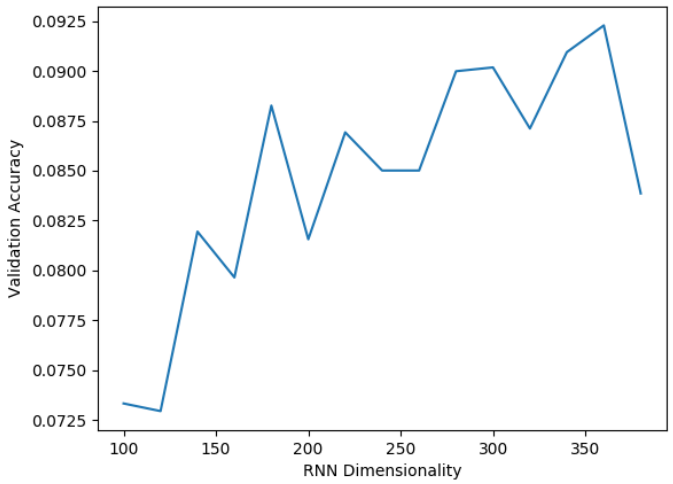
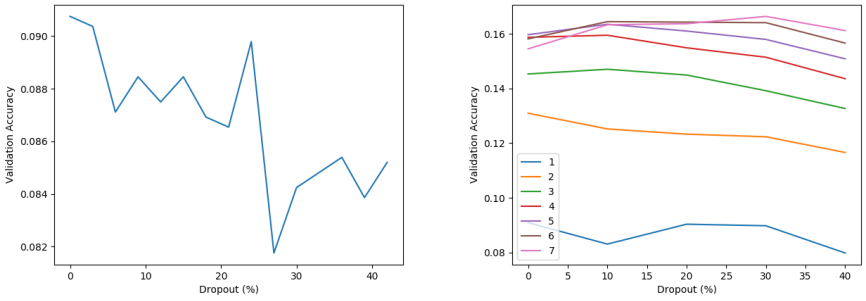


Figure 5.21: Validation Accuracy vs RNN Dimensionality

seconds, meaning that the RNN Dimensionality of the model has seemingly no effect on the time taken to train. As a result, the final model will employ an RNN dimensionality of 360.

Dropout



(a) 1 epoch

(b) Varying epochs

Figure 5.22: Validation accuracy against dropout

Interestingly, figure 5.22a indicates that there exists some negative correlation between the dropout percentage and validation accuracy, which is seemingly counter-intuitive, since the dropout layer serves to specifically increase validation accuracy relative to training accuracy. This could potentially be attributed

to the fact that the dropout layer attempts to reduce overfitting, and that after 1 epoch there is very little room for overfitting. The validation accuracy is often greater than the training accuracy at this point, meaning the model is underfitting instead. To investigate this, the relation between validation accuracy and dropout percentage was tested after each epoch for 7 epochs for 5 models, each with different dropout percentages. The results are shown in figure 5.22b.

It would seem that increasing the number of epochs did mitigate some negative correlation, but it is also evident that not much positive change in validation accuracy was made by applying a dropout layer; There is only a 0.86% improvement attained by applying a 30% dropout compared to 0%. Nonetheless, this came at no cost in terms of time taken to train and test. It can also be observed that a 40% dropout is too high for this model, as it resulted in a decrease of validation accuracy for every epoch. Based on all of this, the final model will have a dropout layer of 30%.

Final Model

Figure 5.23 shows the validation accuracy against the number of epochs for models trained on the 1%, 10% and 100% datasets after hyperparameter tuning. The model trained on the 100% dataset achieved the highest validation accuracy with 21.1% after 7 training epochs, making it just under 4 times more accurate than the baseline accuracy achieved by the naive classifier. As expected, the model trained on the 1% dataset achieved the lowest validation accuracy of 13.9% after 7 training epochs. It is evident that the size of the dataset plays a very large role in the final validation accuracy that is achieved, and there is no doubt that obtaining more data would increase the validation accuracy significantly. There are only 84469 artists as part of this dataset. Thus, even the 100% dataset is a small subset of the available artists and there is much more data that could be obtained in order to increase the validation accuracy of the model. However, the 100% model took approximately 20 hours to train on the development machine, compared to 24 minutes for the 10% model and 1 minute for the 1% model. Clearly, the relation between the time taken to train and the size of the dataset is not linear and would likely diverge massively in a dataset with 1.2 million classes. Consequently, the model trained on the 100% dataset was used as the final model for the application.

5.7.3 Deploying Models

Once the models had been trained and tested, their configurations and parameter weights were saved into JSON and h5 files respectively. The JSON file contained data regarding the model's configuration, including information on the layer types and activation functions thereof. The h5 file consisted of model bias and parameter weight data. The combination of these files allowed the models to be deployed to a large variety of platforms, such as AWS SageMaker. Using the AWS guide on exporting a pre-trained model to SageMaker [20], the models were first exported to ProtoBuf or "SavedModel" format, using both

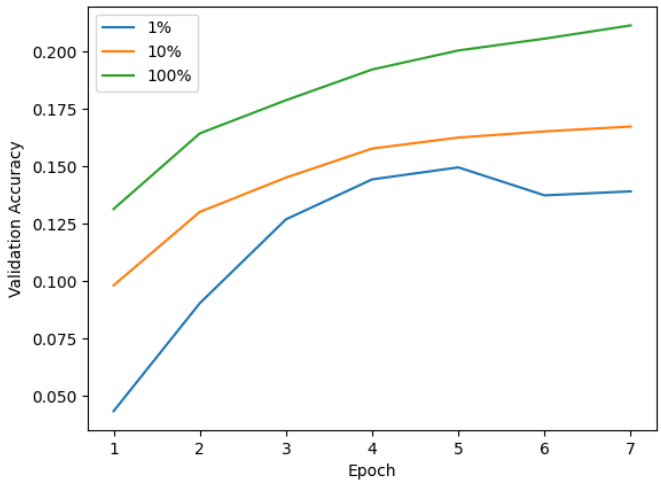


Figure 5.23: Validation accuracy vs Epochs for models trained on different subsets of data

the JSON and h5 files. The ProtoBuf format is essentially a serialisation of the models themselves. The ProtoBuf models were uploaded to an S3 instance which allows SageMaker to create endpoints from them. The endpoints simply invoked the model runtime with the passed input and returned the prediction as calculated by the models. No input encoding/decoding was made at any point by these endpoints, it is expected that any work that needs to be done on the inputs has already been done, and the outputs are precisely that of the model. Consequently, 2 more lambda functions were created to accommodate the encoding and decoding of inputs and outputs for each model, these lambda functions thus served as interfaces between inference requests and the model runtimes.

To implement this for the genre classification model, the tokeniser needs to be loaded inside the lambda function. The keras tokeniser object used when training and testing the model was serialised using the Python pickle package [21]. Once the pickle and keras-preprocessing packages are installed on to the lambda function, the lambda function was able to load and use the tokeniser. While this is largely inefficient and can take up to 8 seconds, it is much more efficient that instantiating a new tokeniser by passing in a dataframe of the train/test dataset. The inverse one-hot-encoding map was used to decode a one-hot-encoded genre seed into a plaintext representation that is usable by the get_recommendations lambda function. Since there are only 126 classes (genres), the actual python dictionary could be copy/pasted from the local training machine to the lambda function, drastically reducing overhead that would arise from serialisation.

For the artist classification model, the matrix of track features needs no altering and can be fed directly in to the model, but the vector of spotify artist ids needs encoding and decoding to and from one-hot-encoded format. Therefore, only the one-hot-encoding map and its inverse are required to interface interaction with the model runtime. However, both maps are 84469 entries large and consequently, the time taken to load the serialised python dictionary is substantial. Fortunately, this is not much of a concern since the entire recommendations process has 25% of the duration of the currently playing track to determine the set of recommendations. The average track is approximately 3 and a half minutes long [22], meaning the entire recommendations process has approximately 52.5 seconds on average.

5.8 Final User Interface

The purpose of the final UI is to satisfy those requirements that the preliminary UI does not. Specifically, the final UI will accommodate the ability to join a lobby through the means of a QR code (requirement 1a), and it will consist of components that are both familiar to users and consistent across the entire UI (requirement 3b).

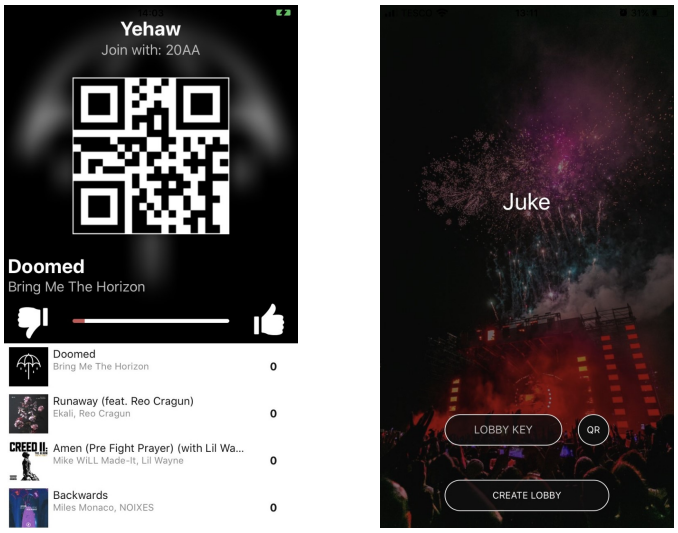
5.8.1 QR Code Capability

With the /join_lobby endpoint already implemented, the development of the QR code system was simple. Since a QR code is fundamentally just a representation of data, a QR code can be made to represent the already-known lobby key (e.g. AB46). Upon scanning such a QR code the lobby key could be extracted and passed into the /join_lobby endpoint.

The QRServer API [23] is utilised by HostLobby.js and InLobby.js so that when a user double-taps the cover image of the currently playing track, the /create-qr-code endpoint is invoked with the lobby key in the body. The response of said invocation is the QR code JPEG which is displayed in place of the cover image as shown in figure 5.24a. Such an implementation enables users who are already in the lobby to easily show the QR code to users who aren't, thus making the process of lobby expansion quick. Consequently, the music playback will become more democratic quicker, which is the foundation upon which the QR code component of requirement 1a was established.

The QR code can be replaced by the cover image of the currently playing track in the same way that it is shown; a double-tap. The react-native-gesture-handler library is used to wrap the cover image in a gesture listener component which is configured to fire after 2 taps.

In order to accommodate the QR code scanning utility, a new UI component was created: Scanner.js. Users can access Scanner.js from the modified Landing.js shown in figure 5.24b. It is worth noting that the "QR" button of Landing.js shown in figure 5.24b becomes a "Join" button once the value of the lobby key text input becomes anything other than the empty string. This



(a) HostLobby.js and InLobby.js (b) Landing.js

Figure 5.24: HostLobby.js, InLobby.js and Landing.js showing QR code support

means that users can still join the lobby through manual input of the lobby key, as is necessary to fully satisfy requirement 1a. With the Scanner.js, the final UI introduces an updated component tree shown by figure 5.25.

Scanner.js utilises the `onBarCodeRead` property of the RNCamera component from the react-native-camera library to pass the extracted QR code data (in this case a lobby key) to a joinLobby method which in turn invokes the /join_lobby endpoint. Upon valid receipt of the request, the InLobby.js component is pushed on to the navigation stack and the user is now in the lobby.

5.8.2 Familiarity and Consistency

User Interaction

User familiarity is important for components that can be interacted with, so that users can quickly deduce the purpose of these components based on previous experiences with other applications, thus reducing confusion. The main components that users can interact with in the developed application are text inputs and buttons. These components were redesigned to mimic a design popular in the Spotify application as shown in figure 5.26, with rounded edges and upper-case text. The result of the redesign can be demonstrated by figure 5.27 which is a screenshot of CreateLobby.js and consists of both a text input (top) and a button (bottom).

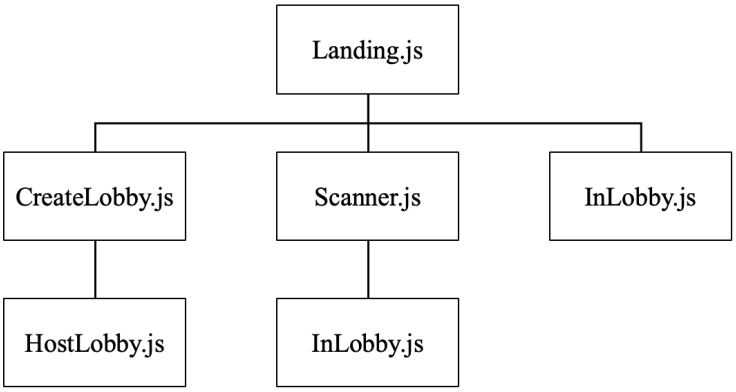
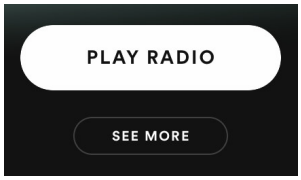
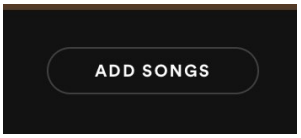


Figure 5.25: Final UI component tree



(a) Example 1



(b) Example 2

Figure 5.26: Spotify buttons

Playback interface

It is important that the UI conveys the playback of the host device in a way that is recognisable for most users, particularly non-host users since it may at first seem foreign and confusing that no music is being played by their own device. Consequently, the Spotify application was once again used as inspiration for making the playback UI familiar. Figure 5.28a shows the Spotify playback UI, currently playing a track called "The Ways (with Swae Lee)" by "Khalid" and "Swae Lee", in a playlist called "Summer 18". Figure 5.28b demonstrates the project application playback UI implementation as part of InLobby.js, where a track called "Cudi Montage" by "KIDS SEE GHOSTS" is being played in a lobby called "CS310". As is evident, these features of the Spotify playback UI have been translated on to the project application, and it undoubtedly makes for a more familiar playback interface than that of the preliminary UI.

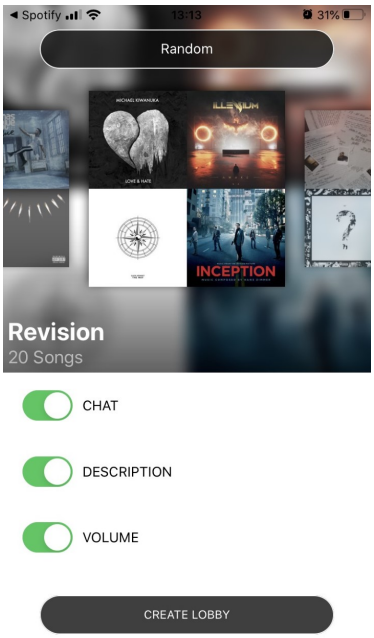
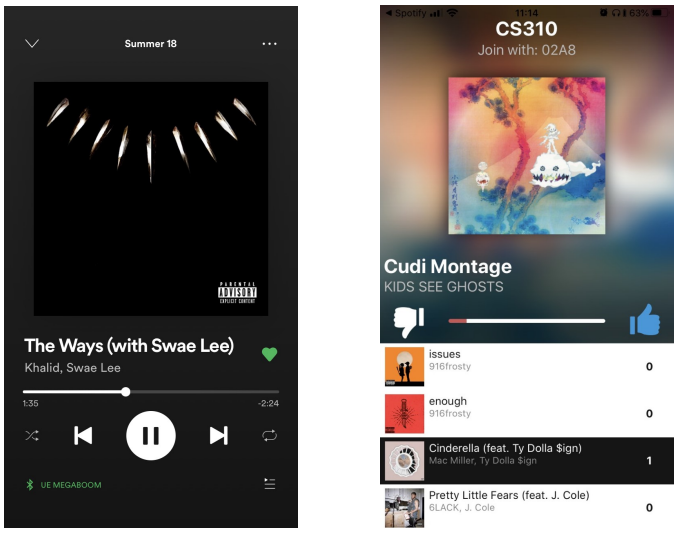


Figure 5.27: CreateLobby.js



(a) Spotify (b) Project application

Figure 5.28: Playback UI comparison

Chapter 6

Testing and Results

6.1 Unit and Integration Testing

6.1.1 Frontend

The frontend and it's internal integration between the native bridging module and the react-native components was tested using the Jest package [24]. Throughout development, Jest was used to continuously snapshot-test the UI, ensuring the underlying tree structure of each UI component did not change unexpectedly. Jest could not be used to unit test the exported macros or internal methods of the native bridging module, because the authorisation process is dependent on user interaction. Therefore, the preliminary UI was manually used to test each of these, which is part of what made the development for the native bridging module so unexpectedly long. Both the problems described in section 5.3 (lazy function evaluation and appRemote connection fragility) were discovered during this manual testing procedure.

6.1.2 Backend

The REST API was first unit tested using Postman [25], where each endpoint would be invoked and the responses monitored for a variety of input cases, focusing mainly on correctness instead of security or latency. If the endpoint gave an erroneous response the lambda function would be inspected alongside the corresponding logs. Through this inspection procedure it was possible to detect and mitigate the problem where any database interaction was not returning a promise, as mentioned in section 5.4. The modularity of the backend API facilitated unit testing incredibly well, as each individual function could be tested and alongside their own individual logs, unaffected of any other API functionality.

The WebSocket API was tested using wscat [26] which allows WebSocket connections to be established and messages sent to various routes over the command-line. Similarly to the REST API testing procedure, each route was

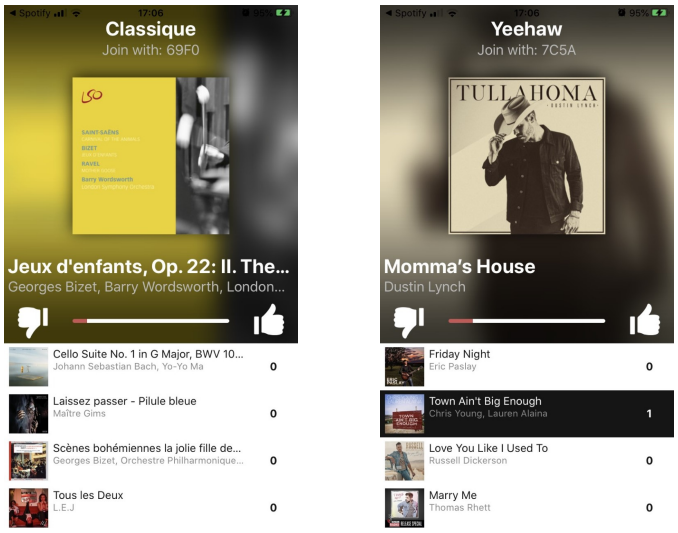
tested individually and checked for correctness. This involved creating "dummy" lobbies using the DynamoDB UI (if lobbies were created using the application, they would be deleted once the application was closed), and attempting to send messages from different dummy users in that lobby. Any incorrect responses meant that the lambda function code was inspected and logs were checked too. In some cases it was revealed that the lambda function was attempting to send responses to users who were not in the lobby, resulting in a HTTP 410 error. Consequently, the exception handling demonstrated in figure 5.14 was written to remove the connection information of any user for which the error is thrown.

The integration between the frontend and the backend was tested using the preliminary UI to invoke react native fetch and websocket requests, for the REST and WebSocket APIs respectively. In doing so, it was possible to test the entire user-lifecycle by building the application to a physical iPhone and running an iOS emulator on the development machine, thus creating two distinct users. At this point, the recommendation system had not been developed and the endpoint simply returned the tracks that made up the base playlist, so that the voting functionality could be tested. The frontend integrated seamlessly with the AWS-hosted backend APIs, seemingly accommodating all valid user-lifecycles.

6.1.3 Recommendation System

Once the final models had been trained and tested, example predictions were performed to essentially unit-test the models. Notably, the genre classification model performed well when any artist names were used as part of the prediction input, and also did particularly well in detecting language. For example, "Beethoven only" would result in classical music being the most probable genre, and "Classique" would result in French being the most probable genre (with classical the second-most probable), as shown in figure 6.1a. "Yeehaw" would infer that the lobby should recommend country music. Clearly, the model was functioning in the way intended. The artist classification model was given numerous example inputs and almost always output artists that were similar to those which were included in the input. It was also notable that artists placed at the top of the input matrix had less importance than those at the bottom, as intended. These example predictions were backed-up by the validation accuracy, which is much more telling of the accuracy of the model than example predictions. The example predictions mainly serve to test how to take the model inputs, format and encode them, input them, retrieve the output, and finally decode them.

Once the example predictions had been performed it was possible to use the same encoding and decoding process in the lambda functions that interfaced between the /get_recommendations endpoint and the hosted models themselves. After these lambda functions had been implemented the integration between the REST API and the hosted models was tested using Postman. The latency for each invocation was massive relative to regular HTTP requests, 8 seconds for genre classification and 15 for artist classification. Nonetheless, they functioned



(a) Classique (b) Yeehaw

Figure 6.1: Example predictions

as expected and could be now invoked from within the /get_recommendations lambda function. Thus, the full /get_recommendations lambda function was implemented and the full user-lifecycle could be tested with a group of users.

6.2 User Acceptance Testing

The project application was built on to the variety of iOS devices of 4 close family members, who were then given the task of creating a lobby and controlling the music playback as a group. Since these were 4 close family members, no proof of ethical consent is required. No specific instructions on how to do so were given in the hope that the UI would be familiar, and it would therefore be clear. The author was observing the group's interaction with the application throughout, taking notes on any difficulties they may face. Once they had concluded their usage of the application, they would report any feedback to the author individually and anonymously.

The group quickly and successfully established a host and created a lobby. The lobby was named "Quarantine Party" by the host, leading to predictions of pop, dance and pump-up music (in that order of probability) by the genre classification model. This was discovered through inspection of the lambda function logs after the test had concluded. Conceivably, the word "party" played more of a role than "quarantine" in this prediction. The host user proceeded by playing one of the recommended tracks, and it was discovered afterwards that this track was not part of the base playlist and was thus inferred by the recommendation system. After which, users began to join the lobby using a

combination of the QR code and the lobby key. All users voted for their favoured next track and at the stage where votes are counted and the next track was deduced, there was a 3:1 majority. The group as a whole gave neutral feedback to the first track of the lobby, meaning the host user's vote-weight remained 1. With no more users joining the lobby, the number of votes for each track paints an accurate description of the feedback that will entail for such track. As was the case, the second track of the lobby attained a 3:1 positive to negative feedback ratio, meaning those users who voted for the track now had a 1.5 vote-weighting. More importantly, the genre classification model was able to give this track preference over the previous one due to the exploitation of the vanishing gradient problem, so that more accurate predictions could be made as to what artist the users would be most likely to listen to next. The next set of recommendations generated contained tracks belonging to the same artist as the well-received track, along with a similar artist to the neutrally-received track. The group continued this cycle of voting and giving feedback for tracks for 3 or so more tracks. At one point, a user closed the application as the playback switched from one track to another, and when they attempted to load the application it crashed. Short after this, the track which was elected was the same track as had just previously been elected, and the UI, specifically the progress bar, demonstrated undefined behaviour as a result.

From the observations of the author, there are plenty of positives to take away from this test. In particular that the recommendation system and user vote-weightings seemed to work well, and users seemed to require no guidance using the UI. Nonetheless, there were problems encountered that were unexpected, such as the application crashing when the user closed it as the lobby state changed, and that the UI responded negatively to the same track being played twice. The individual anonymous feedback reiterated the problems that the author was able to observe, but also included a criticism that it was unclear what will happen when the host user has just created a lobby and selects a track from those recommended, as required.

The problem where what the host user's initial selection entails is a UI problem, since there are no prompts and it is identical to the voting interface. To distinguish the two, a modal was added to the UI upon creation of the lobby that is demonstrated by figure **??**. Included in this modal is a clear prompt telling the user what their action means; By selecting a track, it will be played as the first track of the lobby.

The UI is also responsible for the problem of the progress bar failing to function expectedly when the same track is played in succession, which is a perfectly valid instance of the applications lifecycle. To detect when a new track has begun playing, the progress bar takes in the lobby state as a property, and once this property changes, it is checked to see if the active track is the same as the one before. If it is not, then it is assumed that a new track has begun playing. This is a valid assumption. However, the assumption that if the lobby state is updated and the same track is playing then a new track can't have begun is false. If the same track is to be played twice or more in a row, then the lobby state will update as a consequence of the "next" WebSocket message,

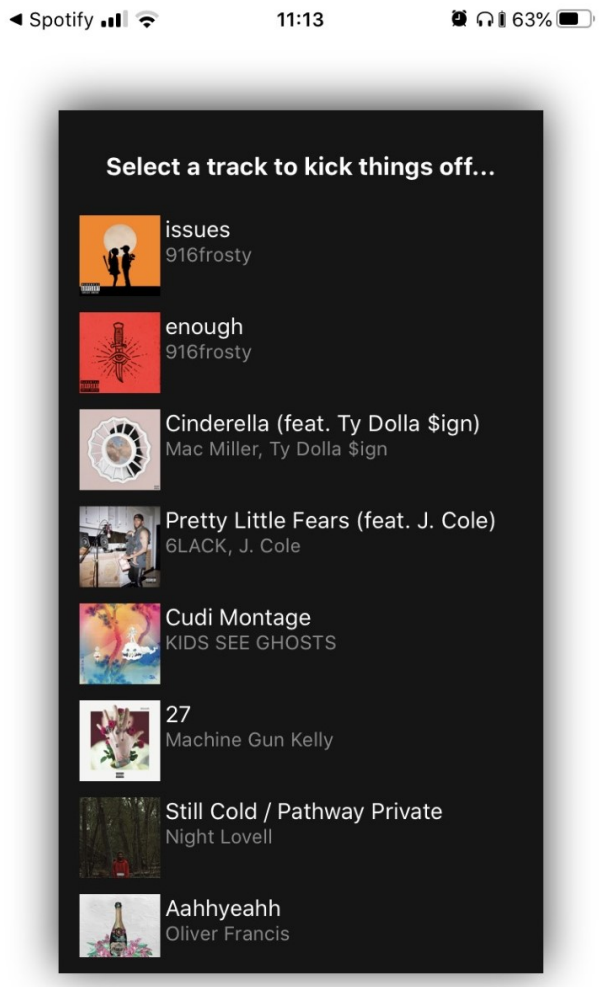


Figure 6.2: Amended Final UI with prompt

but the active track property will have not. Thus, the progress bar will not reset. To correct this, a counter was passed as a property to the progress bar, representing the number of "next" messages the user device has received. If this number changes then a new track must have begun playback within the lobby and so the progress bar resets, even if it is the same track as before.

Unfortunately, in the interest of time, the problem where users cannot close the application during a lobby state change without the application crashing was not fixed as at first glance it is a much more complex problem than the

other two. Such a fix was added to the list of future work.

Chapter 7

Evaluation

7.1 Satisfaction of Requirements

The requirements established in the product backlog were mostly to incentivise the democratisation of music playback. Conceivably, the greater the number of important requirements that have been satisfied, the more democratic the music playback of the environment which the project application provides. The set of requirements was split into 3 sections: User, Lobby and Miscellaneous. labelled 1, 2 and 3 respectively. Here we evaluate how well each requirement is individually satisfied, making reference to tests that support any conclusion.

1a

The host user will be able to create a lobby and invite users to said lobby by means of a key or QR code - This requirement has been unequivocally satisfied. The REST API unit testing showed that it functioned correctly, and no users had any apparent issues joining the lobby with either a key or QR code as part of the user-acceptance testing.

1b

The host user will be able to configure lobby settings before the lobby is created - While the host user is perfectly able to configure all available lobby settings before it is created: Lobby Name, Base Playlist, Volume Control, Track Information, Chat Room. The latter 3 settings have no effect on the lobby itself, since these features have not been fully implemented yet. The ability of users to select a base playlist and lobby name was demonstrated when the users part of the user-acceptance testing did just that. Consequently, the requirement is partially satisfied, though it could be argued that the most important settings of the lobby have been implemented.

1c

The host user will be able to terminate their lobby at any time - At any point within the HostLobby.js UI component, the host user can swipe left to pop HostLobby.js off the navigation stack and as a result the component will dismount. Upon dismount, the /delete_lobby endpoint will be invoked and the lobby will be deleted. This was demonstrated as part of the user-acceptance test and therefore the requirement is fully satisfied.

1d

The host user will be able to authenticate their account with the Spotify accounts service - This requirement is mostly dependent on the function of the native bridging module and the /swap endpoint. Both the native bridging module and the REST API were unit tested, thereby testing the first stage of the authentication procedure components. Moreover, the host user was able to authenticate their Spotify account without any issues. Therefore, this requirement is considered fully satisfied.

2a

Users will be able to join a lobby if they enter the correct key (or barcode) - Similar to requirement 1a, users demonstrated as part of the user-acceptance testing that it is possible to join a lobby but means of a key or QR code. Furthermore, users cannot join lobbies if they enter the incorrect key or scan an incorrect QR code. This was tested as part of the REST API unit testing, when the /join_lobby endpoint was tested for a variety of cases both correct and incorrect. Consequently, whether or not a user joins a lobby is a function of whether or not they enter the correct key or scan the correct QR code. Moreover, a lobby can never be full, since the DynamoDB database has no capacity limit that is going to realistically be reached by users joining a lobby.

2b

Users can only be in one lobby at a time - This requirement is fully satisfied, when a user invokes the /join_lobby endpoint, the lobby key is taken as a parameter and is written to the device item's lobby_key attribute, not appended. Thus, it is impossible for a single device to be in two or more lobbies at any time.

2c

Users can leave a lobby at any time - In the same way that a host user can delete a lobby, a non-host user can leave a lobby; By swiping left. Doing so will pop the InLobby.js UI component off the navigation stack and consequently the /leave_lobby endpoint will be invoked, which will remove any data linked to the device.

2f

Users can choose to thumbs-up or thumbs-down the current track - As proven by REST API unit tests including various test cases for the /thumbs endpoint, and the user-acceptance testing, this requirement is fully satisfied. Users who partook in the user-acceptance testing reported no difficulty in giving feedback for the current track, and most users did so for most tracks.

2g

Each user has one vote for the next track - While user votes may be worth different amounts, each user has only one vote. Using wscat [26], it was possible to unit test the vote route and it was evident that any old vote was deleted before any new vote is made. As a result, the requirement is fully satisfied.

2h

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 track they have voted for - The user vote weight adjustment shown in equation 4.2 is a function of the average rating for a track, and is applied to all users who voted for that track. This is mathematically true and is demonstrated by the user-acceptance testing where the vote weights of each user became that which the equations in section 4.6.2 would suggest. Therefore, this requirement is fully satisfied.

2i

A user cannot obtain a vote weight of zero - The vote weight adjustment shown in equation 4.2 completely prevents any vote weight from becoming zero. At minimum, a user's vote weight will half each track, but will never mathematically become zero. Therefore, this requirement is completely satisfied.

3a

The backend must be scalable - The code that composes the lambda functions which make up the backend API is stateless, so lambda can start as many copies of a function as needed, and can dynamically allocate capacity to match the rate of incoming events [14]. As a product of this, the requirement is fully satisfied.

3b

The user interface will be responsive and familiar to users - The UI was designed so that each UI component is fit to the screen of the device, using the screen width and height in the calculations of the components width and height. In theory, this means that the UI is responsive. This can be confirmed by the fact that the variety of iOS devices that were used in the user-acceptance test all

had the same UI, and functioned expectedly despite the change in resolution and aspect ratio. While it is difficult to objectively evaluate the similarity between the application UI and that of Spotify, the application UI was based on features from the Spotify UI, and most users found no difficulty working out what each component did in the user-acceptance testing. in consequence, this requirement is fully satisfied.

3d

The recommendation system will give lobbies recommendations on the next tracks to play and will be used as a tie-breaker when two or more songs have equal votes - The recommendation system currently functions by recommending tracks to be elected based on a variety of factors. This part of requirement 3d is fully satisfied. However, the tie-breaking feature has not yet been developed, and is included in the list of future work. As a result, this requirement is only partially satisfied.

3e

The recommendation system will make use of objective user-preference analysis to infer suitable recommendations - This requirement is fully satisfied. The artist classification model takes its inputs ordered by track weighted ranking, which is an attempt to approximate user-preference. Consequently, those tracks which had the highest weighted ranking carry more weight into the classification process and therefore are more likely to have an impact on the set of final recommendations.

3g

The recommendation system will be hosted on an external application server - The recommendation system is fundamentally composed of two models, which were trained locally but exported to external S3 instances. Consequently, this requirement has been fully satisfied.

7.1.1 Summary

In total, 15/21 (71.4%) of the requirements are fully satisfied, 2 (9.5%) are partially satisfied, and 4 (19%) are not satisfied and are instead on the list of future work. Those that have been not satisfied are of the lowest importance, and contribute to the democratisation of music playback the least.

7.2 Project Management

7.2.1 Time Management

Seen in table 7.1 is an accurate timetable of the tasks that have been encountered over the course of this project. It is worth noting that task 10 is yet to

No.	Task	Date
1	Write specification	01-Aug-19
2	Project planning	03-Aug-19
3	Begin development	05-Aug-19
4	Submit specification	09-Oct-19
5	Submit progress report	25-Nov-19
6	Oral presentation	02-Mar-20
7	Finish development	15-Mar-20
8	Test application	30-Mar-20
9	Fix any failed tests	10-Apr-20
10	Submit final report	11-May-20
11	Complete further work	15-Jul-20

Table 7.1: Actual timetable

be complete, as this task takes place after the submission of this report. In comparison to the proposed timetable (see 3.2), the key changes are that the development took slightly longer than expected, and that the Covid-19 situation has meant that the deadline for submission of this report was extended by 2 weeks. As a result, there was more time to finalise any development, test the application, and then fix any problems that become apparent from such testing. The decision to deadline the oral presentation preparation for the 02-Mar-20 was well made, since this is the earliest possible date and the author was also randomly allocated this date, meaning that no change in schedule was required. The time pressure peaked at both the point of the progress report, and the weeks leading up to the oral presentation. Otherwise, time was well managed and the project development was paced well.

7.2.2 Risk Management

All 3 of the contingency plans devised in section 3.4 were utilised in an attempt to mitigate any negative impact that may occur to the project's development.

Large time constraint

The time constraint was judged to have become too large at the point of the progress report and the lowest importance requirements were appended to the list of future work, and requirement 3f (or whatever it was) was removed from the product backlog. Consequently, the author was perceivably given more time to focus on the important aspects of the project, those requirements which contribute most to the democratisation of the project.

Development machine breaking down

After the oral presentation, the development machine broke and was rendered completely unusable. Thankfully, this was good timing as the only component

that was scheduled to be developed further after the presentation was the final UI. However, the broken machine would drastically hinder the testing process if github was not used for source control, as the application code would be unattainable. Thankfully, a replacement machine was found quickly, and the code was pulled. After approximately one day of re-configuring the development environment, the project application could be built from the new machine to accommodate any further development or testing. In consequence, the contingency plan to utilise source control in case of such an event could have saved months of re-development.

Legal issues with public lyrics

The legal issue of displaying the lyrics was avoided by amending requirement 2f so that instead of displaying lyrics, track information is displayed. This worked as well as intended in that there was no longer any legal issue, but the resultant implementation was deemed unreliable. In turn, further development would be required to design and implement a reliable solution, and the time constraint was too large, so the amended requirement was placed on the list of further work.

7.2.3 Methodology

The scrum methodology fit the development of the project well, and its dynamic nature accommodated the amendment of the product backlog suitably. Moreover, the sprint cycles allowed the author to frequently demonstrate any new features to the key stakeholder and receive any feedback that they may have.

Chapter 8

Future Work and Conclusions

8.1 Future Work

4 of the requirements have been moved from the product backlog to the list of future work, which will be completed after the submission of this report.

8.1.1 Chat Room

Another WebSocket route will be developed to accommodate full-duplex messaging functionality within each lobby, and each message will be stored as part of the lobby item so that all messages will be deleted upon lobby termination. The /join_lobby endpoint will be adapted to retrieve the lobby messages so that any new users can view past messages, and the preliminary ChatRoom.js UI component will be extended to employ the design of the final UI. Users will be able to toggle the chat room and as a result requirement 2e will be fully satisfied.

8.1.2 Track Information

The implementation which invokes the /get_description endpoint when track information is required and then facilitates the user's need to be able to show and hide said information, is already in place. Further development will be made to the endpoint so that the information attained is much more reliable. In order to do so, it may be necessary to depend on a different API, which itself is less dependent on user input which is the root cause of the unreliability in the current implementation. As a result of this proposed further development, requirement 2f would be fully satisfied.

8.1.3 Volume Control

The volume control system will not control the volume state of the device itself, but more the volume state as part of the Spotify application. Therefore, a new method will have to be created as part of the native bridging module, which will interact with the playback API through the native appRemote object, enabling a new exported macro which can be invoked from HostLobby.js, controlling the playback volume. In alignment with the ethos of the project, the host user will be given no direct interface to invoke such method, and it will only be invoked if the users of a lobby vote for it. This would introduce a new component of lobby state; The volume control vote state. Resultantly, a new WebSocket route will need to be implemented to facilitate a new stateful communication across each lobby, invoked whenever a user votes to increase or decrease the volume. Such a solution would result in the complete satisfaction of requirement 2k.

8.1.4 Deployment to the App Store

In an effort to make the application available to the public, it will eventually be deployed to the iOS app store in order to satisfy requirement 2l. Before this is done however, the application and API security would have to be rigorously tested, and more extensive testing must take place to ensure the stability of the application. Moreover, it is intended that in order for the application to be ready for deployment, most items on this list of future work will have been completed too. Consequently, the application still has a long way to go before this requirement is satisfied.

8.1.5 Tiebreaker

Part of requirement 3d specifies that the recommendation system should act as a tiebreaker so that if two or more tracks attain equal vote scores, the elected track will be the one which the recommendations system ranks the highest. The track which ranks the highest will conceivably be the one that most closely matches the inferences from the 2 classification models. Again, however, this could result in a tiebreaker. In this scenario it would be sufficient to randomly select the track from among those that achieve the highest rank.

8.1.6 More Training Data

It is verifiably the case that for both classification models, the number of training examples has a positive effect on the performance with respect to accuracy when given unseen data. However, the time taken to train each model is disproportionate to the change in sample size; It takes much longer than twice as long to train a model on a dataset of twice the size. After the submission of this report, when there is seemingly no time pressure, this issue is conceivably less problematic. Consequently, more data will be collected using the data-scraping Python scripts, and the models can be retrained with the same parameters, but

with more data, and then deployed in the same way. As a product of this, the recommendation system will be better equipped to give users accurate recommendations that they are more likely to vote for.

8.1.7 Change of lobby state with an inactive application

It was made clear during the user-acceptance testing that the application could not handle the lobby state attempting to change while the application was inactive. This is because the WebSocket API will attempt to send a "next" message to the client, but the client will not be able to receive this message since the application is inactive. As a result, the API will interpret this as a HTTP 410 client gone error, and consequently remove the user from the lobby. Then, once the application is made active again, the user has either HostLobby.js or InLobby.js on top of their navigation stack, but is not registered to be in the lobby according the backend. In order to fix this, the logic behind what happens when a user makes the application inactive or when the API receives a 410 error must be reviewed, since it is unrealistic to expect users to keep the application active at all times.

8.2 Conclusions

This project encompasses a large variety of technologies, all combined to produce an application which perhaps goes further than any other in way of the democratisation of music playback. Users can easily join lobbies and vote for their favoured tracks. Host users are given the ability to quickly connect their Spotify account to the application and thereby sacrifice their complete playback control to the preference of the lobby. Whichever track receives the greatest vote score is queued and automatically played on the host device. The recommendation system produces relevant recommendations modelled on both the dynamic combined user-preference of all users in the lobby, and the nature of the social setting, which is inferred from the lobby name.

There is a large absence of public un-hashed data which can be used to help build music recommendation systems, making the development of systems that employ modern and conventional collaborative-filtering techniques such as matrix factorisation much more arduous. In consequence, recommendation systems that wish to be applied to real applications must adopt different approaches, such as the combination of a natural language processing model and a recurrent neural network.

Cloud computing services such as AWS offer plenty of support in the development of multifaceted applications, where various features can be deployed within a single AWS environment and thereby integrated seamlessly. Moreover, serverless event-driven APIs provide increased modularity over traditional monolithic architectures, at the price of increased temporal overhead.

Plenty of future work needs to be done before the application is deemed sufficient for release by the author, and the target of 15-Jul-20 may need to be

extended. But overall this project is seen as a success by the author, who has learned 4 new languages/frameworks, how to train, test and deploy machine-learning models, how to create cloud-hosted applications, and how to develop software using a scrum methodology.

Chapter 9

Bibliography

[1] J. Constine, "Spotify is building shared-queue Social Listening." Accessed: May 5, 2020. Available: `https://techcrunch.com/2019/05/31/spotify-social-listening/`.

[2] L. Bernard, *Festify*. (2018). Available: `https://festify.rocks/`.

[3] *OutLoud*. (2.0.9). Available: `https://outloud.dj/`.

[4] "The Scrum Guide," in *Software in 30 Days*, 2015.

[5] Project Management Institute, *A guide to the project management body of knowledge (PMBOK® guide)*. 2008.

[6] Apple, *Apple Music API Documentation*. (1.0). Accessed: May 5, 2020. Available: `https://developer.apple.com/documentation/applemusicapi`.

[7] Spotify, *Spotify iOS SDK Documentation*. Accessed: May 5, 2020. Available: `https://developer.spotify.com/documentation/ios/quick-start/objective-c/`.

[8] C. W. Chen, M. Schedl, P. Lamere, and H. Zamani, "Recsys challenge 2018: Automatic music playlist continuation," in *RecSys 2018 - 12th ACM Conference on Recommender Systems*, 2018.

[9] H. Zamani, M. Schedl, P. Lamere, and C. W. Chen, "An analysis of approaches taken in the ACM Recsys challenge 2018 for automatic music playlist continuation," *ACM Transactions on Intelligent Systems and Technology*, 2019.

[10] L. J. Tardón, I. Barbancho, A. M. Barbancho, A. Peinado, S. Serafin, and F. Avanzini, "16th Sound and Music Computing Conference SMC 2019 (28–31 May 2019, Malaga, Spain)," *Applied Sciences*, 2019.

[11] "Company Info." Accessed: May 5, 2020. Available: https://newsroom.spotify.com/company-info/.

[12] Spotify, *Spotify Web API Documentation*. Accessed: May 5, 2020. Available: https://developer.spotify.com/documentation/web-api/.

[13] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, 2014.

[14] "AWS Lambda FAQs." Accessed: May 5, 2020. Available: https://aws.amazon.com/lambda/faqs/.

[15] Genius, *Genius API Documentation*. Accessed: May 5, 2020. Available: https://docs.genius.com/.

[16] P. Lamere, *Spotipy*. (2.12.0). Available: https://spotipy.readthedocs.io/en/2.12.0/.

[17] "English Words Dataset." Accessed: May 5, 2020. Available: https://github.com/dwyl/english-words.

[18] M. Pichl, E. Zangerle, and G. Specht, "Towards a Context-Aware Music Recommendation Approach: What is Hidden in the Playlist Name?," in *Proceedings - 15th IEEE International Conference on Data Mining Workshop, ICDMW 2015*, 2016.

[19] Google, *Keras Functional API Documentation*. Accessed: May 5, 2020. Available: https://keras.io/getting-started/functional-api-guide/.

[20] Amazon, *Deploy trained Keras or TensorFlow models using Amazon SageMaker*. Accessed: May 5, 2020. Available: https://aws.amazon.com/blogs/machine-learning/deploy-trained-keras-or-tensorflow-models-using-amazon-sagemaker/.

[21] Python, *pickle - Python object serialization*. Available: https://docs.python.org/3/library/pickle.html.

[22] M. Tauberg, "Music is Getting Shorter." Accessed: May 5, 2020. Available: https://medium.com/@michaeltauberg/music-and-our-attention-spans-are-getting-shorter-8be37b5c2d67.

[23] GoQR, *QRServer API Documentation*. Accessed: May 5, 2020. Available: http://goqr.me/api/doc/.

[24] Facebook, *Jest*. (26.0). Available: https://jestjs.io/docs/en/getting-started.

[25] *Postman*. (7.23.0). Available: https://www.postman.com/product/api-client/.

[26] *WSCat*. (4.0.0). Available: https://www.npmjs.com/package/wscat.