Santa Clara University
Scholar Commons

Computer Science and Engineering Senior Theses | Student Scholarship

6-6-2016

Live

Samuel Kujovich
Santa Clara University

Griffin Cook
Santa Clara University

Tyler Selewicz
Santa Clara University

Follow this and additional works at: http://scholarcommons.scu.edu/cseng_senior

Part of the Computer Engineering Commons

Recommended Citation
Kujovich, Samuel; Cook, Griffin; and Selewicz, Tyler, "Live" (2016). *Computer Science and Engineering Senior Theses.* Paper 61.

This Thesis is brought to you for free and open access by the Student Scholarship at Scholar Commons. It has been accepted for inclusion in Computer Science and Engineering Senior Theses by an authorized administrator of Scholar Commons. For more information, please contact rscroggin@scu.edu.

SANTA CLARA UNIVERSITY

Department of Computer Engineering

I HEREBY RECOMMEND THAT THE THESIS PREPARED
UNDER MY SUPERVISION BY

Griffin Cook, Sam Kujovich, Tyler Selewicz

ENTITLED

Live

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

BACHELOR OF SCIENCE
IN
COMPUTER SCIENCE AND ENGINEERING

Yuhong Liu — 06/03/2016

Thesis Advisor — date

N. Ling — 6/6/2016

Department Chair — date

Live

by

Samuel Kujovich, Griffin Cook, Tyler Selewicz

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science in Computer Science and Engineering
School of Engineering
Santa Clara University

Santa Clara, California
June 3, 2016

Live

Samuel Kujovich, Griffin Cook, Tyler Selewicz

Santa Clara University
June 3, 2016

ABSTRACT

Music streaming applications that do not require listeners to actually own the music they listen to are quickly become the most popular and most cost effective way to listen to music. These applications however are limited in their capabilities for playlist collaboration, specifically for real time collaboration. When an app does provide a way for users to collaborate on playlists, the users must be friends who have explicitly granted each other access to edit playlists. Our solution aims to improve on existing applications by allowing users to collaborate on music playlists in real time based on their immediate location.

Live is a mobile jukebox allowing people to connect with those around them over a common love, music. We pull music from existing services so users can continue to listen to all of the music they are accustomed to, but we also provide users with better ways to discover new music. In addition to providing a platform for real time location based playlist collaboration, the application recommends songs based on what people in a user's immediate location are listening too. Our application's key features will change the way that users both find and share their music.

Contents

1 Introduction **6**
1.1 Problem Statement . . . 6
1.2 Background . . . 6
1.3 Solution . . . 7
1.4 Requirements . . . 7
1.4.1 Functional Requirements . . . 7
1.4.2 Nonfunctional Requirements . . . 7
1.4.3 Design Constraints . . . 8

2 Design **9**
2.1 Use Cases . . . 9
2.2 Activity Diagram . . . 12
2.3 User Interface . . . 13
2.4 Technologies Used . . . 18
2.5 Architectural Design . . . 19
2.6 Design Rationale . . . 20

3 Project Management **21**
3.1 Testing . . . 21
3.2 Risk Analysis . . . 22
3.3 Developmental Timeline . . . 23
3.4 Ethical Issues . . . 23
3.4.1 Issues Regarding Content Creators . . . 23
3.4.2 Issues Regarding Users . . . 23

4 Conclusion **24**
4.1 Future Improvements . . . 24
4.2 Lessons Learned . . . 24

List of Figures

1.1 Functional Requirements and Importance. 7
1.2 Nonfunctional Requirements and Importance. 8
1.3 Design Constraints and Importance. 8

2.1 Use Case Diagram. 9
2.2 Activity Diagram. 12
2.3 Home Screen 13
2.4 Login Screen 14
2.5 Now Playing Screen 15
2.6 Top Songs 16
2.7 User Profile Screen 17
2.8 Data-Centric Architectural Model. 19

3.1 Risk Analysis Table 22
3.2 Developmental Timeline. 23

List of Tables

2.1 Play a Requested Song 10
2.2 View Most Requested Songs in a Geographical Location 10
2.3 Build a Collaborative Playlist 11

Chapter 1

Introduction

1.1 Problem Statement

When people want to listen to music these days, many look to mobile applications to listen to playlists or just stream individual songs. Users currently utilize streaming services, which allow them more access to music without having to actually own the songs they would like to listen to. Many of these services are public facing, allowing users to share playlists created in their applications with friends, family or the general public. Although this model is convenient, it does not allow other users to edit playlists, and due to the contracts set up between streaming services and artists, popular artists occasionally do not host their music on these services. With the exponentially expanding rate at which music is being produced, it is becoming harder and harder for users to find good songs and create an enjoyable playlist.

1.2 Background

There are multiple existing solutions to this problem, but each has its own quirks that we hope to improve upon. One current solution is the music streaming application Spotify. Spotify looks to bring restricted content owned by record labels to users through online streaming. There are two tiers to the service, Spotify Free, which costs no money, but occasionally serves advertisements, and Spotify Premium, which is paid for through a monthly subscription.The problem with this product is that there is no way to easily collaborate on playlists, meaning it is difficult to have multiple people contribute to a specific playlist. Another solution is Google Music, an online radio streaming service. Users can select songs, genre, or artists, and hear a playlist that relates to their selection. Google generates "recommended for you" playlists and stations based on artists in your music library, provides a listing of the most popular songs in their service, and offers users the ability to create their own playlist. Again, the major problems in this service relate to playlist creation and additions. All playlists must be created in the web interface that Google provides, and although one can share a playlist, others cannot add to it. Apple Music is the most recent solution to this problem. Apple looks to join the content streaming space by allowing users to create playlists, see recommended playlists, or even listen to radio stations based on genre. Like Spotify, Apple Music is also paid for via a monthly subscription. It currently provides no way for users to collaborate on playlists. Another very popular application, Soundcloud, also looks to take on this challenge. Soundcloud allows users to stream content, create playlists, interact with others, and even post songs. It is a free service, that is available through a mobile app and a web app. Soundcloud attempts to provide a "Trending" songs feature, which should list popular songs at the time, but the feature seems to be broken as the songs featured in that portion of the application rarely seem to update. Playlists created by other users are difficult to find, and the mobile application is not intuitive and often does not function properly. The balance between making money and providing functionality has truly handcuffed the user experience with many of these applications.

1.3 Solution

Our project aims to improve upon the current solutions by adding features to allow users to collaborate based on immediate location and demand for individual songs. While a user plays music from the mobile application, other users will be able to suggest songs to the playlist if they are in its general vicinity. The more a song is suggested, the more likely it will be played next, which makes requesting a song very easy if enough users want to listen to it, but also makes it so a single user cannot abuse his or her right to request songs. Since the playlist can be created by all of its listeners, the application will eliminate the need of having a person in charge of handling song request manually. Our solution aims to pull music from existing services, but because it will be a time-consuming process for a user to find new music, we aim to recommend new music to users based on what people in the same places as them are listening as well. We believe that our solution will make both playlist creation and music recommendation significantly easier and quicker.

1.4 Requirements

The objectives of our project will be referred to as requirements. A functional requirement is something our application will do. A non-functional requirement defines the manner in which our project will meet a functional requirement. A design constraint will limit how the project can be created.

1.4.1 Functional Requirements

The functional requirements are listed in Figure 1.1, and are ranked by how necessary the team views them to the final project. The importance to the final product is ranked on a scale of 1 to 10, with 10 being crucial for our final project to succeed, and 1 being a stretch goal for our project.

Requirement	Importance (1-10)
The application will play any requested songs it can locate	10
The application will find a requested song from public facing third party streaming services	10
The application will allow certain playlists to be accessible and editable by certain users through security settings	10
The application will allow users to collaborate in real time on playlists in the their area	10
The application will provide users with new music suggestions based on location	8
The application will prioritize song order based on how many people are requesting each song.	8
The application will authenticate users via Facebook	8

Figure 1.1: Functional Requirements and Importance.

1.4.2 Nonfunctional Requirements

The nonfunctional requirements are listed in Figure 1.2, and are ranked by how important the team views each requirement to our final product.

Requirement	Importance (1-10)
The application will be user-friendly	10
The application will be responsive	10
The application will be intuitive	9

Figure 1.2: Nonfunctional Requirements and Importance.

1.4.3 Design Constraints

The design constraints are listed in Figure 1.3, and are ranked by how important the team views each requirement to our final product.

Design Constraint	Importance (1-10)
Native iOS application	10

Figure 1.3: Design Constraints and Importance.

Chapter 2

Design

2.1 Use Cases

Use cases give a list of steps defining the interaction between an actor and the system to complete some goal. They give any preconditions for a specific use, the flow of the events to complete that functionality, any postconditions after the use has been completed, and any error conditions or alternative flows that a user may encounter.

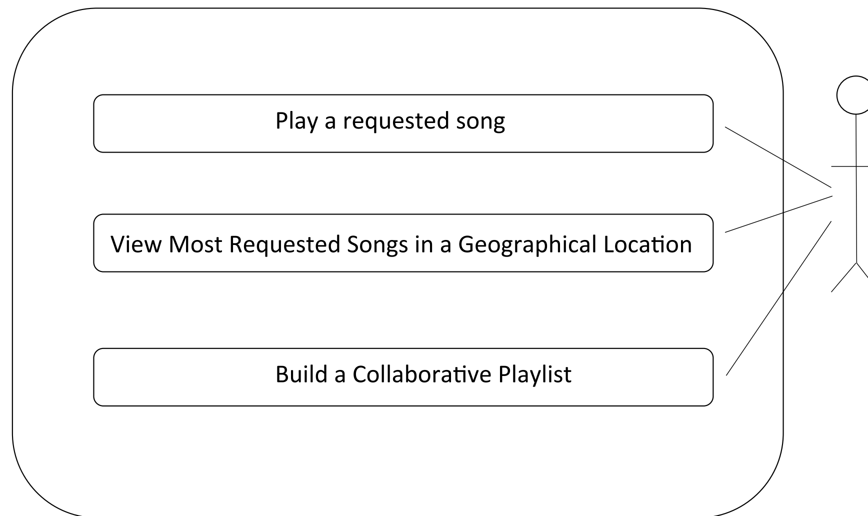

Figure 2.1: Use Case Diagram.

Our use case diagram, shown in Figure 2.1, outlines 3 particular use cases that describe the solution. They are Play a requested song, View Most Requested Song in a Geographical Location, and Build a Collaborative Playlist, located in tables 2.1, 2.2 and 2.3 respectively.

Components	**Details**
Actor	User
Goal	To Play a Requested Song
Preconditions	The user has an account The user has a connection to the internet User must be included in the group that can edit a playlist
Postconditions	New Song Added to Queue.
Steps	User submits title and artist of desired song System monitors number of times a song is requested As the number of requests for the song increases, the high priority is placed on the song.
Exceptions	The Song will not be played if it has been recently (within one hour).

Table 2.1: Play a Requested Song

Components	**Details**
Actor	User
Goal	To View Most Requested Songs in a specified Geographical Location
Preconditions	The user has an account The user has a connection to the internet User has application User has Location Services Enabled
Postconditions	List of most requested songs in an area are displayed
Steps	Open Application Select "Top Tracks" Tab Select Genera of Choice
Exceptions	No Requested Tracks near current location Location Services turned off

Table 2.2: View Most Requested Songs in a Geographical Location

Components	Details
Actor	User
Goal	To Build a Collaborative Playlist with a Select Group of People
Preconditions	The user has an account The user has a connection to the internet The user has the applicaiton
Postconditions	New Playlist Added Successfully
Steps	Open Application Select "Create New Playlist" Submit title of new playlist Select other users that will be able to request songs Notification sent to all users who can collaborate on the playlist
Exceptions	N/A

Table 2.3: Build a Collaborative Playlist

2.2 Activity Diagram

An activity diagram is a graphical representation of workflows (organizational and computational processes) of stepwise functions and activities. Activity diagrams contain support for concurrency, iteration and choice and, in short, show the overall flow of control. They are regarded as a type of flowchart. The activity diagram below illustrates what the process of using our application is like for the user.

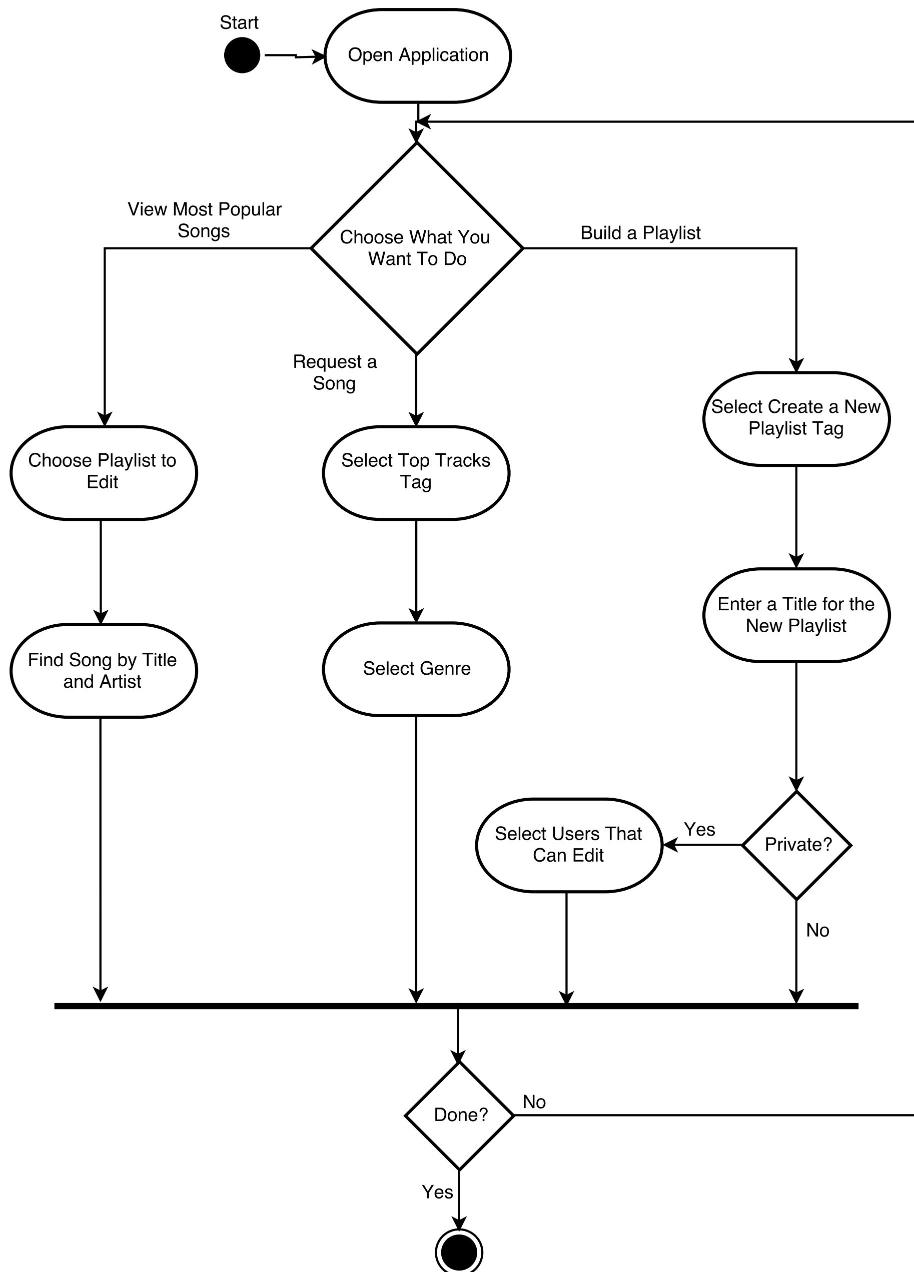

Figure 2.2: Activity Diagram.

2.3 User Interface

Our system takes the format of a native iOS application that the user can download from the Apple App Store. The user will then login and will be presented with a main screen. From this the user will have the option to play any of the local playlists or navigate to one of the apps other screens. One of the tabs allows a user to see their own playlists that they have created. Another tab allows them to see and control what is currently playing. The last two tabs allow a user to view the top 10 songs in their area and to view their profile and settings.

The home screen of the application is shown in Figure 2.3. The application is split into screens that are accessable via the navigation bar across the bottom of the screen. The users can select the view that is affiliated with their goal. The login screen is shown in Figure 2.4.

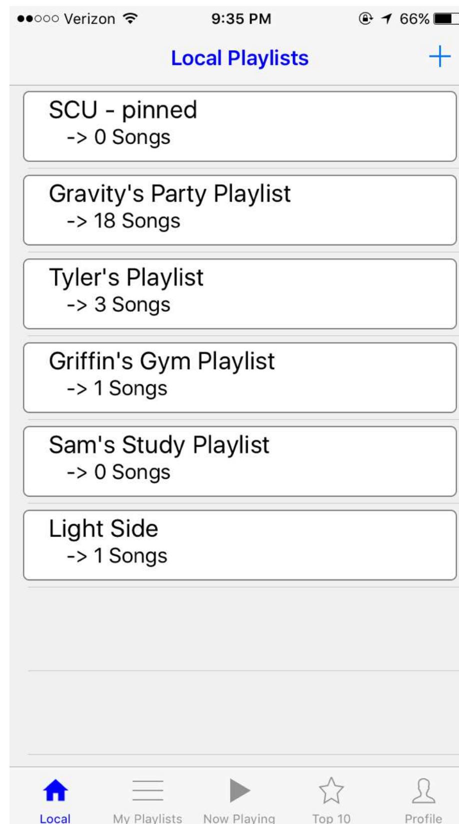

Figure 2.3: Home Screen

●●●●○ Verizon 1:27 PM 10%

Live

Welcome to Live

To prevent fake users we ask you to log in
using your Facebook account. Live will never post
anything using your Facebook.

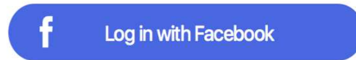

Figure 2.4: Login Screen

Figure 2.5: Now Playing Screen

Once the user is connected to a playlist and listening, they can select the Now Playing tab, which displays the current song's album art and provides information about the playlist. This can be seen in Figure 2.5. Figure 2.6 shows the top songs screen where users can see what the most popular songs in their area are. Users can also go to their profile tab where they can view their own profile. This screen is shown in Figure-2.7.

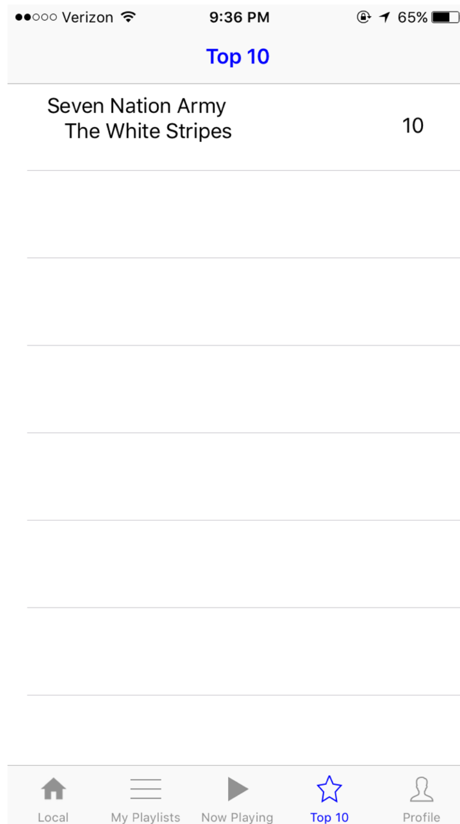

Figure 2.6: Top Songs

Figure 2.7: User Profile Screen

2.4 Technologies Used

For the backend of our system, we used Parse, a service run by Facebook that essentially acts as a frontend for a MongoDB database and a metrics system, that allows for easy control and easy communication between any client and that server. Our frontend is an iOS application that we used the Apple iOS SDK to create. We also used Github as a version control system and Xcode as our integrated development environment (IDE) during the development process.

2.5 Architectural Design

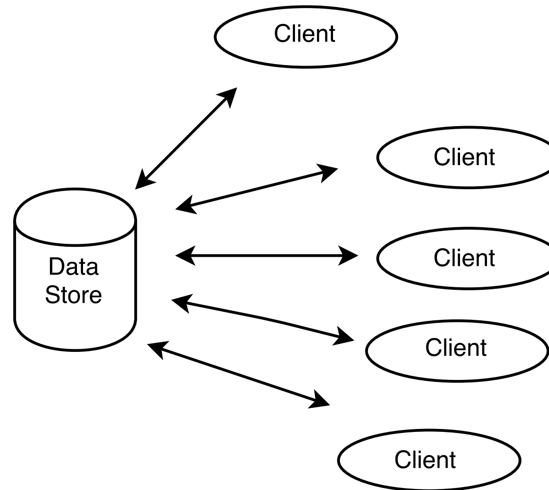

Figure 2.8: Data-Centric Architectural Model.

This section identifies the architectural design model our system falls under. Choosing an architectural design is important because changing the architecture during the design and implementation phases is costly and time consuming.

Our project will be modeled after a Data-Centric Architecture as seen in Figure 2.8. There are two pieces to the system. The data store (server) which will host our database and process information, and the client which will display the information from the sever. These two components will be connected through reads and writes from the client to the data store to enable users to indirectly interact with the server. This is important for our application, as the system is user driven, meaning the playlists change as users change. The system must be able to effect change quickly, and effectively. A benefit of using this model is also to allow for many clients to connect and disconnect to the data store, and each client does not need to know that the others exist. The problem with this is that all clients are connected to the data store and thus if there is a problem with the data store then there is a problem with the entirety of the system.

2.6 Design Rationale

Our system is intended to provide a user-friendly experience when attempting to listen to music and create playlists from multiple public-facing music streaming services. We chose to create our own interface to access these services to reduce the need to switch applications to listen to a different song.

Our team decided on creating a native iOS application instead of a mobile web application for a variety of reasons, namely the goal of reducing drain on battery life. By making a mobile web application, the mobile web browser would act as an extra layer between our application and the system, causing more function calls to be made to achieve the same goal on a native application, which can cause immense battery drain in some cases. We feel that music applications should not be too heavy on the battery of a mobile phone, so choosing a native application seemed clear to us. On top of this, one of our team members has extensive experience in iOS development, which outweighed the lack of experience our team had in the intricacies of web development, which helped make our decision very easy.

We then looked at how we would like to store playlist data. To provide information to multiple clients efficiently, we decided the best approach would be a client-server interface. Our user data, metadata, and playlist information is all stored in a service called Parse, which provides an easy-to-use frontend for a MongoDB database and a metrics system. By using a service instead of creating our own server, we mitigate a bunch of the risk involved with maintaining our own system, making sure we have efficient and secure networking code, and scaling our own system, as Parse handles every one of these for us. Parse also provides extensive mobile SDKs that simplify the communications between the client applications and their service, so we were able to use their iOS API as part of our client side.

We then decided on what streaming services we would use. The first one we started with was Soundcloud. It has the friendliest HTTP API that allows any third party service to access it and play songs, but we soon thereafter discovered inconsistencies in their search feature that left as longing for more options in addition to Soundcloud. Due to the service's recent change from solely a free-to-listen model that included occasional advertisements to a two-tiered model that has a free tier where you can listen to most songs, but not all, and a paid tier in which you can listen to every songs, the data being returned by their HTTP API has been censored to account for said changes.

From here, we thought about what we could use to fill in the cracks in our search issues, and looked towards Spotify. With its library of high quality music, we knew we would get access to a quality API, but we noticed one glaring issue: they will only stream songs to our application if the user is signed into a Spotify Premium account, which is their paid tier. We decided that this option is still a good idea, as it will allow the user to support their service while being able to interact with the music in the more convenient way that our application provides.

For the users who choose not to use Spotify Premium, though, this glaring hole in our search feature still exists, and that is why we ended up choosing Youtube as our last music source. The quality is going to be lower as much of the data coming in from a Youtube video is not actually the audio, so that would be a notable discerning factor in the situation.

Chapter 3

Project Management

3.1 Testing

Testing is a very important piece of the software engineering process. This is the portion in which the developers, company, and clients use the application with the goal of finding bugs.

For a mobile application there are a few important bugs to look out for. The first is application crashing, which is when an application crashes frequently when trying to execute. Another major bug is application incompatibilities, which occurs when an application does not work on all devices it is designed for (i.e. iPad, iPhone, iPod or iOS 7,iOS 8, iOS 9). Due to the nature of our application being connected to Facebook, there is the need to protect personal information that is being granted. This leads to another vulnerability in the app, security. This is where a hacker would be able to steal a user's information. The fourth major bug is memory leakage, which is where blocks of allocated memory are no longer used by the program and can lead to crashes.

To combat against these bugs we took numerous steps. The first was to implement automated testing. These automated tests looked to target each SDK that we used within the application, as well as simulate the app on multiple devices. This helped with increasing the efficiency of our baseline tests.

Our next phase was unit testing using the unit testing SDK that is provided through Apple, XCTest, which is built into XCode. This allowed us to run seamless unit tests with low overhead and high compatibility with the coding software we used, XCode.

Finally we looked to test our User Interface by physically deploying the application to numerous devices and attempting to use and break the application. The first stage of this was to deploy the app on our own devices, simulating white box testing. The second was to deploy the application to other devices, where users did not know about the code, or have technical skills. This allowed us to accurately replicate the experience a user will have.

3.2 Risk Analysis

The risk analysis table outlines the risks that we recognized we might encounter through the entire development process. Risks are listed in descending order of Impact (Probability * Severity) and each has a pre-determined list of foreseen consequences. The last column in the risk analysis table lists our mitigation strategy for the listed risks. Our risk analysis table is shown in Table 3.1.

Risks	Consequences	Probability	Severity	Impact	Mitigation Strategy
Bugs	Incorrect songs played/displayed. Further testing and more programming/error checking needs to be done	0.8	8	6.4	Go through test cases slowly and in unison, documenting any anomalies witnessed. Document bugs to ensure that no bugs are reoccuring
Illness	Re-Evaluation of timeline. Work redistribution	0.5	7	3.5	Sync up to ensure memers are on the same page. Side-by-side coding sessions to understand each other's code
Market Misunderstanding	Working on a product that is not needed.	0.5	9	4.5	Constant market research. Constant analysis of competitors in same space to avoid building something that already exists
Technological Inability	Lost time learning needed technical skills.	0.2	3	0.6	Identify mandatory needed skills early. Shuffle responsibilities as needed

Figure 3.1: Risk Analysis Table

Over the course of developing our application we did encounter two of the risks which we recognized in our risk analysis. First, we had problems with technological inability. We had to change our initial plan for the backend because we did not have enough experience to build everything from scratch quickly enough. Luckily since we discovered this early enough in the process, we were able to change our design to something more doable. We did not think it was very likely that we would encounter this problem, but luckily we did recognize the possibility and planned accordingly. Second, we had problems with bugs. This problem was much more expected, but that does not mean it was easy to deal with. Even with careful documentation of anomalies or bugs in the application, it was still sometimes very time consuming to fix certain bugs that arose.

3.3 Developmental Timeline

The developmental timeline illustrates the schedule we followed to develop our system from start to finish and is shown in Figure 3.2.

Fall
Winter
Spring
Problem Statement
Design Document
Backend on Parse
Develop iOS views
Integrate UI with Parse
Testing
Debugging
Optimization
Acceptance Testing

Figure 3.2: Developmental Timeline.

We spent the entirety of Fall Quarter defining the problem we wanted to solve and designing our solution to that problem. Winter Quarter we began to implement the design starting with the backend and then moving the frontend user interface. The end of Winter Quarter and beginning of Spring Quarter were spent testing and finalizing the application's implementation.

3.4 Ethical Issues

As engineers we have an obligation to consider how the products we create affect society. We must work by a standard of ethics to ensure we benefit society as a whole.

3.4.1 Issues Regarding Content Creators

One of the biggest ethical issues in our project involves how we provide music to our users. All of this music is owned by someone else, and that person should be able to benefit from their music being played. The easy solution to this is playing music through premium accounts for services like Spotify, because this ensures the user has already paid for the rights to listen to the musics. If a user does not have a premium account, then artists can still generate money through advertisement revenue. If we were to stream music to users in such a way that did not play the advertisements, the owners of the music would not get the money that they deserve. When we built our application we had to make sure that each of the ways that we access music sources are both legal and ethically right.

3.4.2 Issues Regarding Users

The biggest ethical issue regarding our users is ensuring their privacy. Our application needs to collect certain user data to function properly, including a users location. If we do not transmit and store this data in a way that ensures users' privacy and anonymity, someone with malicious intent could potentially gain access to sensitive user data. When a user uses our application, they are trusting us to be responsible with their information, and we have an ethical obligation to protect their privacy.

Chapter 4

Conclusion

4.1 Future Improvements

There are several improvements and new features that could make our application better in the future. The first is increased security settings. Right now our application does not have a way to control who has access to edit each playlist. If a user creates a collaborative playlist, anyone in their area will be able to request new songs to it and vote on songs. Future development could add functionality so that a user can customize who has privileges to make changes to their playlists.

Another future improvement could be adding Apple Music as a music source. While Apple Music is still not as popular as Spotify and Soundcloud, it does offer some artists' music that the other music sources do not.

Adding a feature so that users can download playlists could be another improvement to the application. Spotify for example allows users to save music for offline listening so they can enjoy their music even when they do not have a data connection. Currently users cannot use our application unless they are connected to the internet.

Finally, future development could add functionality to recommend new music to users. Right now the application can suggest new music based on what is popular in the area, but these song recommendations are the same for every user in a location. With a large enough user base, we could get enough user data to implement a recommendation system to give recommendations tailored specifically to each user.

4.2 Lessons Learned

The biggest lessons we learned throughout the development process of this application were to prototype early, focus on core functionality first, and simplify the design wherever possible. Starting out we had several big ideas for features for our application that we tried to implement, and in doing so we ended up with a terribly buggy product that could not even play music, which was supposed to be our most important feature. We learned that it is much better to focus on the most important things first, and then build off of that to add secondary features.